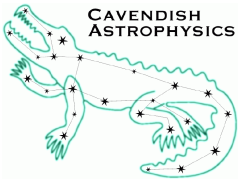
MRO Delay Line

Trolley Software Functional Description

The Cambridge Delay Line Team



Cavendish Laboratory
Madingley Road
Cambridge CB3 0HE
UK

Objective

To describe the design, implementation and functionality of the trolley software.

Scope

This document forms part of the documentation for the delay line final design review. It describes the design, implementation and functionality of the software that runs on the central processing unit (CPU) of the delay line trolley. Also included are brief descriptions of the interfaced hardware and the development environment and their impact on the software design.

This document should be read in conjunction with the document "Trolley Electronics Design Description", which contains further information on the interfaced hardware and a description of the firmware in the on-board programmable multi-axis controller (PMAC).

Contents

1 Introduction

The delay line trolley CPU is a single board computer embedded in the electronics bay of the trolley. It performs many tasks:

- Receipt of commands from the workstation and streams of data from the metrology system and shear camera via a wireless ethernet link.
- Control of actuators and motors on the trolley in order to introduce a carefully controlled optical path difference into the delay line.
- Measurement of on-board sensors in order to close servo loops and provide real time feedback and telemetry data for the user.
- Transmission of telemetry data to the workstation for logging and user feedback via a wireless ethernet link.

Three test programs have been produced to carry out these tasks. They run under Linux on the CPU. The programs are:

- *Alacarte*, a user program that communicates with various devices and performs higher-level processing and coordination.
- *Dmm48*, a Linux kernel module for communication with the trolley's analog I/O boards. This module is a heavily modified version of a module written by Diamond Systems Corporation (the board vendor).
- *Pmac*, a Linux kernel module for communication with the PMAC. This module is a modified version of a module written by Michael Ashley of the University of New South Wales.

This document describes each of these programs, and the environments in which they are developed and run, in more detail.

2 Development Environment

The operating system used is Linux (2.6.11), chosen for the absence of licence fees, previous development experience by the authors and open source code availability. The development language is C because it is the preferred language for Linux kernel module development and because it is familiar to all programmers in the delay line team. *Alacarte* is written in an object-oriented way to facilitate porting to C++ should that prove desirable.

Development occurs on a Debian Linux "testing" (currently "lenny") personal computer. As the development computer and target computers have incompatible processor architectures (Intel x86 and ARM-compatible RISC respectively) code must be cross-compiled to run on the trolley CPU. Furthermore, it is necessary to have access to the CPU's kernel source code to compile the trolley modules. A development kit from the CPU vendor (Arcom) is used, which includes the source code and the *arm-linux-gcc* cross-compiler.

For *alacarte*, the *glib* library is used as a framework. This library is more well known as the foundation for the *gnome* graphical user interface but can also be used independently, as it is here. *Glib* sets up an event loop which can be interrupted by "watches", such as the arrival of data from the analog to digital section of an analog I/O card, or an internal timer signal. The use of *glib* in this way makes the trolley program fully event-driven.

The two kernel modules, on the other hand, use the Linux 2.6 kernel application programming interface. They are also event driven, reacting to events supplied by the kernel.

Once software has been compiled, it is transferred to the CPU via the ethernet port.

3 Runtime Environment

The computer is an Arcom Viper V1I6 single board computer (this will be a V2I4 in the production trolley). It features an Intel PXA255 400MHz ARM compatible processor, five serial ports, an ethernet port and an I^2C interface. The bus is PC104 compatible, allowing the computer and various peripheral boards to be stacked together into a robust, compact unit. The bus is used to communicate with the motor controller and two analog input/output cards, while power is derived from a PC104 compatible switch-mode power supply.

The processor has an ARM compatible instruction set. The source code is mostly independent of this, but in several respects it is not:

- The ordering of words is reversed to that of Intel x86 systems. This becomes an issue when data is transferred between the trolley and other computers, as frequently occurs over the network during normal operation. The messaging protocol used does not handle this automatically and it must be dealt with explicitly.
- There is no floating point coprocessor so all floating point computations are done in software and consequently are expensive.
- An on-board timer is used in a system-dependent way to accurately trigger an interrupt that polls the analog I/O cards checking for data.

There are 64 megabytes of RAM present, ample to store Linux and the trolley programs, and flash memory is used to emulate a 16 megabyte hard drive (this will be 32 megabytes on the production cart) upon which the programs are stored when the trolley is powered down.

Finally, the CPU clock is kept synchronised with clocks in the other computers in the delay line system using the NTP protocol so that telemetry data can be compared between systems. The CPU receives commands and sends real-time and archival data via a custom protocol developed in-house for this purpose. Additionally an engineer can log in to the CPU using *ssh* via the network to perform maintenance or debugging tasks.

4 The User Program (Alacarte)

An overview of the user-level program architecture is illustrated in Figure 1. Apart from the use of the *glib* framework, the most influential factor in the overall design has been robust network management due to the difficulty in physically accessing the trolley while in service.

4.1 Terminology

Before proceeding, a note on terminology. The custom network protocol developed for the delay line distinguishes between four types of message:

- Commands. These are orders and may have several parameters attached. They require acknowledgements.
- Command data. These are streams of numbers sent over the network to close servo loops that span more than one computer.
- Status data. These are messages that describe the current state of the sender and are sent at 10Hz or faster. The intention is to give the user real time feedback of what is happening at the various devices, although the information is also logged for future analysis.
- Telemetry data. These are messages sent at XXXHz or faster that contain arrays of data not deemed necessary for real-time viewing. The sample rates are usually higher than for status messages, and many sequential values can be sent at once. Telemetry messages are logged for future analysis.

For further information on the protocol please see the documents "Messaging Protocols Document", which discusses the protocol in detail, and the appendix of "XXX", which discusses how the computers initiate connections.

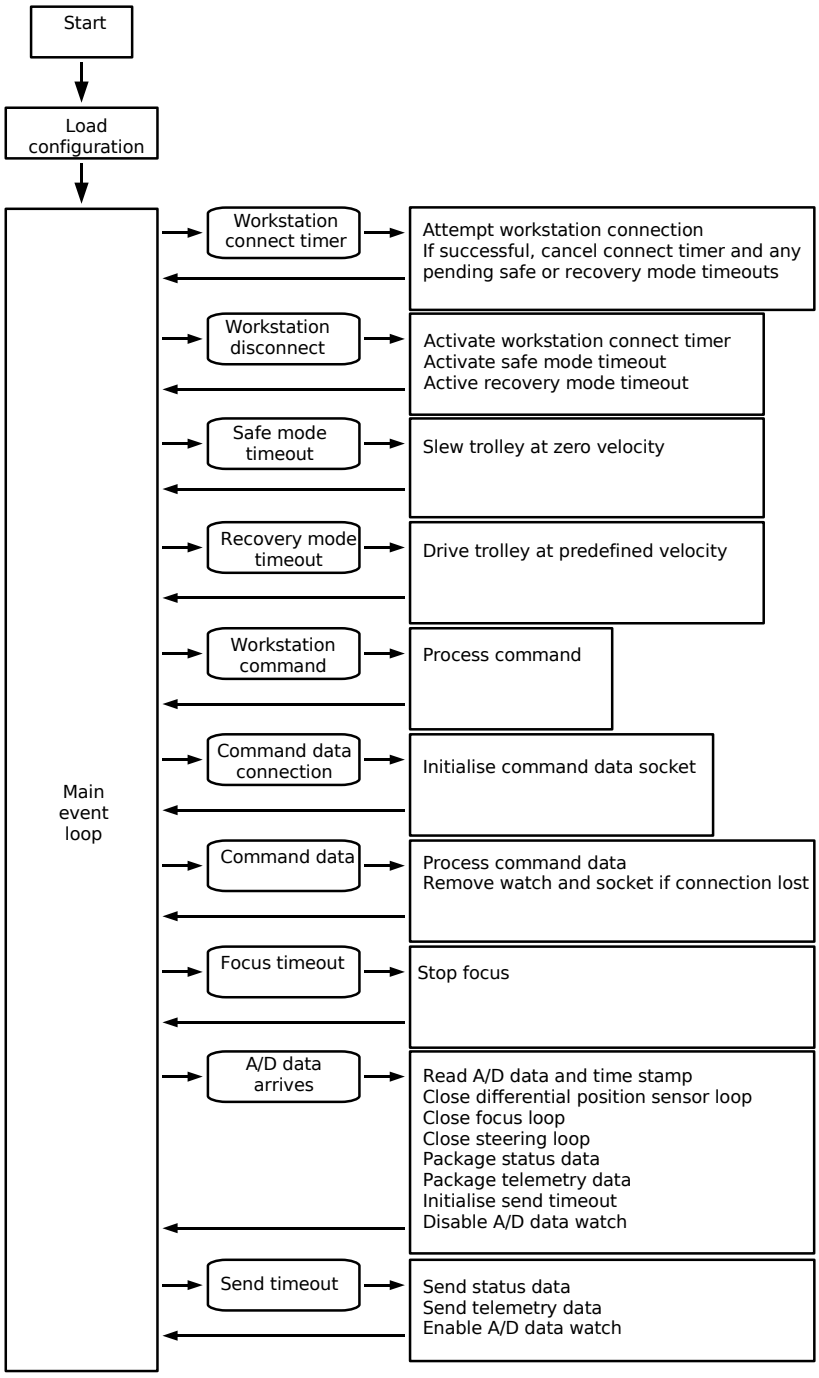


Figure 1: Overview of the user program architecture.

4.2 Initialisation

The program is stored on the local onboard flash disk. On startup, it loads a configuration file which is also stored locally. This allows a user to set many program parameters without recompiling the code. Configureable options include:

- Network configuration such as an identifier, the address of the workstation, and which network port numbers to use.
- In the event of a network failure, timeouts in seconds before attempting to reconnect, enter a safe mode, or enter a recovery mode.
- Paths to devices used by the program.
- Sample rates.
- Analog to digital parameters. For each of the 16 input channels on each of the two analog I/O boards, the user can specify the channel name, whether it is to produce status or telemetry messages (or both), the sample rate, gain and offset values to convert the incoming bits into the physical quantity they represent, and the physical units. Hence the program behaviour can very easily be changed to reflect wiring changes to the analog inputs.
- Drive and steering motor servo parameters.
- Motor parameters. The PMAC reports drive and steering motor positions and speeds to the CPU, which can then be processed in a similar way to the analog to digital parameters discussed above.
- Secondary mirror tip-tilt servo parameters.
- Secondary mirror focus servo parameters.
- Digital to analog parameters. These are configureable in a similar way to the analog to digital parameters discussed above.
- Digital outputs. Output lines can be assigned here.

The program also initialises all internal objects and sets up some network connection timers.

Once initialisation is complete, the program enters the main event loop, which as described above is managed by *glib*. It waits there for various events to happen. The possible events are described in more detail below.

4.3 Workstation connect timer

During initialisation, a timer is set to cause an event once every five seconds (this value can be changed in the configuration file). On this signal the program attempts to initiate a connection to the workstation via the network. If the connection is successful, the program is able to receive commands from the workstation and the timer, along with any pending contingency timers (see Subsections 4.5 and 4.6 below) is cancelled. If no connection is made, further connection attempts will be triggered by the timer until one is successful.

4.4 Workstation disconnect

If the connection to the workstation is broken, perhaps by a network or workstation problem, this event causes the timer described in Subsection 4.3 to be reinitialised, so that the trolley computer will reconnect to the workstation automatically once the problem is fixed.

Two other one-shot timers are also initialised here. These are a safe mode timeout and a recovery mode timeout, described in Subsections 4.5 and 4.6 below. The periods are as specified in the configuration.

4.5 Safe mode timeout

The safe mode timeout event, which occurs when the workstation connection has been broken for a given number of seconds, causes the trolley to stop (more precisely, to slew with zero velocity). The timeout period can be set in the configuration.

4.6 Recovery mode timeout

The recovery mode timeout event, which occurs when the workstation connection has been broken for a given number of seconds, causes the trolley to slew with a given velocity. The timeout period and the velocity can be set in the configuration.

This event handler was written so that the trolley would automatically move to a pipe end for access in the event of a lengthy network interruption. However, it has since been decided that the ability of the trolley to move independently without warning was dangerous and this velocity is now set to zero. However, the code has been left in in case it is needed.

4.7 Workstation command

When connected, the workstation can send commands to the CPU to change the behaviour of the trolley. When a packet arrives on the ethernet port it is firstly checked to see if it is from the workstation, is intended for this particular trolley, and is an allowed trolley command. If it passes these tests it is parsed and action is taken according to the command. In some cases, the command requires data from the the analog I/O boards and the only immediate action is to set a flag so that the command is remembered when the next set of data arrives from them (Subsection 4.8).

The following commands are recognised:

- SteeringOn: Turn on the steering servo.
- SteeringOff [angle]: Turn off the steering servo and move the steer wheel to [angle].
- TipTiltOn: Turn the secondary mirror tip-tilt servo on.
- TipTiltOff [X] [Y]: Turn the secondary mirror tip-tilt servo off and tilt the mirror to angles [X] and [Y].
- FocusPos [position] [timeout]: Move focus to [position] and stop on arrival or on [timeout], whichever happens first.
- FocusOffset [distance] [timeout]: Move focus [distance] from current position and stop on arrival or on [timeout], whichever happens first.
- DirectSlew [velocity]: Slew the trolley at [velocity]. Overrides command data velocities sent from metrology system.
- DirectSlewOff [velocity]: Slew the trolley at [velocity] but allow the metrology system to override.

4.8 Command data connection

Alacarte maintains a listen socket that waits for incoming command data connections on the allocated port number (as defined in the configuration file). This event is triggered when an external command data source (either a shear camera or the metrology system) attempts to connect to the socket.

The routine then tries to allocate a dedicated socket for communication with the command data source. If successful, it adds a *glib* "watch" to trigger a command data event whenever data arrives on the new socket (Subsection 4.9).

A maximum of five command data connections are allowed (the number can be changed in the configuration file).

4.9 Command data

This event is triggered whenever data arrives on a command data socket. Expected sources are the metrology system and whichever shear camera is allocated to the trolley's delay line, and the data from them is used to close the metrology and shear loops on the trolley.

The data is firstly checked to see if the source, destination and contents are allowed and consistent. If it passes these tests it is parsed and action is taken according to the contents. In some cases, data from the the analog I/O boards is also needed before processing and the only immediate action is to set a flag so that the command data is remembered when the next set of data arrives from the boards (Subsection 4.11).

The following command data is recognised:

- Slew [velocity]: Slew at [velocity], unless a direct slew command is in force.
- Track [velocity]: Track the metrology demand position while moving at [velocity].
- TipTiltOffset [X] [Y]: Move tip-tilt mirror [X] and [Y] from current position.

4.10 Focus timeout

This event is triggered a predefined time after a focus command has been given and tells the focus drive to stop. The timeout period is set within the focus command (Subsection 4.7). This is to protect the focus drive if for some (possibly mechanical) reason it never reaches its destination.

The timeout is cancelled if the focus reaches its destination before the timeout expires.

4.11 Data arrival from analog to digital converters

This event is triggered by the arrival of raw data from the analog I/O boards. This happens every 0.1 seconds as defined by a sample clock on one of the boards. Hence all the actions described in this Subsection occur at a frequency of 10Hz.

On receipt of the event, the program firstly reads 0.1 seconds worth of data into a local buffer. There are 32 channels of data (16 for each of the two boards), each of which is 16 bits wide and currently sampled at 5kHz. The program also gets an estimate of the time the signals were sampled from the device driver.

The most recent 32 readings are then used to close servo loops:

- If the trolley is currently tracking, the value read from the catseye differential position sensor is added to the requested motor velocity that the metrology system sent. The result is sent to the PMAC, which controls the motor. Hence the motor position is continually adjusted to maintain a cart position underneath the catseye.

- If the focus servo is on, the value read from the focus encoder is used to calculate the required speed and direction of the focus drive to get the focus to the required position. This is converted into a drive voltage by a digital to analog converter, which is then applied to the focus drive.

- If the steering loop is closed, the value read from the trolley inclinometer is used to calculate a steering angle. This angle is then sent to the PMAC, which controls the steering motor. The aim is to keep the trolley level.

  The algorithm is as follows: If the tilt is more than a value specified in the configuration, apply a proportional- integral (PI) servo.[1] Otherwise, if the trolley has just crossed the ideal tilt value then it is steered straight. Otherwise, the steering is set one motor step away from straight in the correcting direction as indicated by the inclinometer. This algorithm is designed to minimise the use of the steering stepper motor, which draws significant current while stepping.

The most recent 32 readings and status information (both local and from the PMAC) are then bundled up into a status message to be sent to the workstation.

The entire block of readings from the analog to digital converters is then converted into telemetry data. For each channel, the configuration file specifies whether a conversion into telemetry is required and a sample rate. If a given channel is to be sampled at less than the maximum rate, it is downsampled. This reduces the floating point load on the CPU and network traffic. Local flags and PMAC data are also bundled into the telemetry message.

The data is not sent immediately, as there is then a possibility that all the interferometer trolleys would decide to send their data at once, causing congestion over the network and on the workstation. Instead, each trolley is allocated periodic timeslots with respect to system time when it is allowed to transmit. These timeslots are offset between trolleys so that no two can transmit at once, provided that their system clocks are all synchronised with NTP. For a given trolley, the local system clock time is inspected and a *glib* timer is armed so that it generates an event during the next available timeslot (Subsection 4.12). The timeslot offset is defined in the configuration file and is currently set to 0. It is possible that the MROI's gigabit ethernet network and the workstation will be able to handle these bursts of traffic, in which case this section of code will become redundant.

[1] Scaling of the proportional and integral components can be changed in the configuration file. The integration component has not been needed so far and is scaled to zero.

Finally, the analog to digital data arrival event is cancelled so that it does not trigger before the existing data is sent.

4.12 Send timeout

The send timeout is timed to trigger during a timeslot allocated for transmission of status and telemetry information. Accordingly they are both sent to the workstation. After dispatch the analog to digital data arrival event is rearmed, ready for the next chunk of data to arrive from the analog I/O boards.

5 The Dmm48 Kernel Module

The *dmm48* kernel module is responsible for communication with the analog I/O boards. These are a pair of Diamond Systems Diamond-MM-48-AT PC104 boards. Each has 16 analog inputs, 8 analog outputs, four digital lines (input or output), 4 optocoupled inputs and 8 relays. They were selected for the wide variety and number of inputs and outputs offered, allowing excellent flexibility during the trolley design phase.

The module communicates with the boards by reading from and writing to assigned addresses in the PC104 bus memory space. It appears to the user as a range of character device files, each of which supports a subset of the board functionality.

The module is an extensive rewrite of a module provided by Diamond Systems Corporation. The rewrite was necessary because:

- The original module contained proprietary Intel x86 binary code and hence could not easily be ported to run on the CPU's ARM processor.
- A PC104 bus interrupt line incompatibility was discovered between the analog I/O cards and the CPU. The I/O cards are designed to support multiple interrupts per line while the CPU is not. The result is that the CPU cannot detect interrupts generated by the I/O cards. Hence the software had to be modified to support polling of the I/O card status.

5.1 Initialisation

The device files needed by *dmm48* are created at boot time by a script which then loads the module itself. The module accepts a list of up to eight analog I/O board base addresses as parameters, although it is unlikely in practice that more than two will ever be used.

Once installation is complete, the user can configure the analog I/O cards using the control interface (see Subsections 5.3 and 5.4 below).

5.2 Analog to digital conversions

Each board has 16 analog inputs, each of which is sampled in turn by a single on-board 16 bit analog to digital converter. On board 0, a free-running clock is set to trigger these scans at a user configureable rate (normally 5kHz). The clock output is wired into an external clock input on board 1 which is thereby slaved to sample at the same rate and time as board 0. On each board there is a 2048 sample first-in-first-out (FIFO) buffer that temporarily stores the digitised data before it is read by the module.

Both FIFOs set a flag when they are at least half full. The module checks the flags every 6.4ms until they are both set, and then reads out 1024 samples from each. The check is triggered by an internal timer interrupt on the CPU that is set up for the purpose, and at the above data rate is sufficient for a buffer to fill by 512 samples.

As the data is read out, it is interleaved by board number, so that each 32 sample chunk contains 16 samples from board 0 and 16 samples from board 1, all sampled by the same trigger. After interleaving, the data is placed in a kernel FIFO buffer. When the amount of data in this buffer grows to be 0.1 seconds' worth or more, the current wall clock time is noted and the module's read function is woken up. This signals *Alacarte* in user space that the data is available for reading, and *Alacarte* can then read the data and the clock time and process them as described in Subsection 4.11. From the user's point of view, all the channels from both boards are all read from a single device, currently `/dev/ad0`.

The polling method of determining the FIFO status is inefficient. Furthermore, when a sample is read out of the FIFO there is no way of knowing how old it is for the purposes of time stamping, and the time stamp could in fact be up to 6.4ms late. It would be much more efficient to trigger a hardware interrupt when the FIFO was exactly half full and note the time then. However, as noted above this option is not available due to differences in interrupt implementation between boards.

A method has been devised to correct for the time stamp problem: an internal timer interrupt gets triggered once every system clock second and changes a logic level. This logic level is then attached to one input channel on an analog I/O board and it thereby accurately "marks" the data stream. An input channel ("Sync") has been allocated for this purpose but the signal has not yet been implemented.

5.3 Other channels

A range of other, much simpler, devices are provided for reading from and writing to the analog I/O boards. A byte read from the optocoupler device indicates the state of the optocoupler inputs. A pair of bytes written to a digital to analog device causes a voltage representing the value to be output from the equivalent pin. Similarly, bytes written to the relay and digital I/O devices change the states of their outputs.

A control device is also provided for more complex tasks. Reading from it provides the contents of all of a board's registers, while writing to it allows the contents of any of those registers to be changed.

5.4 The IOCTL interface

The module makes a variety of useful housekeeping commands available via the IOCTL interface:

- IOCTL_DMM48_RESET: Resets the boards.
- IOCTL_AD_CONF: Configure the analog to digital converters based on the contents of a passed data structure.
- IOCTL_DMM48_GET_VERSION: Returns the version number of the module.
- IOCTL_DMM48_GET_SAMPLE_T: Returns the sample time.
- IOCTL_DMM48_GET_NUM_BOARDS: Returns the number of boards in use.
- IOCTL_DMM48_FREE_IRQ: Free the timer interrupt request line used to poll the hardware FIFOs.
- IOCTL_DMM48_WRITE_BYTE: For the control device, writes a byte to the specified register in the I/O address space.
- IOCTL_DMM48_READ_BYTE: For the control device, reads the contents of a specfied byte in the I/O address space.
- IOCTL_PXA255TC_START_INTERRUPTS: Starts the polling timer interrupt.
- IOCTL_PXA255TC_STOP_INTERRUPTS: Stops the polling timer interrupt.

6 The Pmac Kernel Module

The *pmac* kernel module is modified from code provided by Micheal Ashley of the University of New South Wales. The main changes are a port from the Linux 2.4 kernel to 2.6 and introduction of the ability to poll a PMAC rather than relying on interrupts, as the PC104 version of the PMAC has no interrupt support.

Like *dmm48*, *pmac* is loaded at CPU boot time by a script. It accepts just two parameters: the hardware base address, and a polling interval.

The PMAC hardware communicates with the CPU via the PC104 bus using a text based protocol: characters are read from and written to a single address in the CPU's PC104 address space. Hence from the user's perspective, *pmac* is much simpler than *dmm48*, because one simply writes bytes to or reads bytes from a single device. Effectively the PMAC behaves like a text terminal.

The PMAC contains a 2048 byte buffer for outgoing messages. This is checked by the module for pending characters at the polling interval (currently 1ms). When characters are detected, they are read into a ring buffer, ready for reading by *alacarte* during its 10Hz read cycle. Similarly, data written to the module by *alacarte* goes into a ring buffer until it is read by the PMAC hardware.

A single IOCTL command is also available, PMAC_IOCGSTATUS, which returns some PMAC status information to the user. It is not used in *Alacarte*.