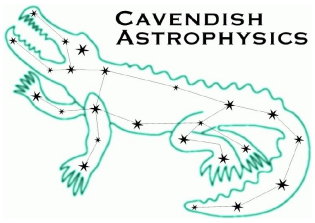
# MRO Delay Line

## Socket Initialisation Protocol for Delay Line Computers

***E. B. Seneta***
***bodie@mrao.cam.ac.uk***

***rev 0.2***

***13 October 2006***




Cavendish Laboratory
JJ Thomson Avenue
Cambridge CB3 0HE
UK

## Objective

To describe the protocol by which network connections are to be made and broken between delay line computers.

## Scope

This document describes how to make and break network connections for the purpose of sending and receiving delay line command, command data, status and telemetry messages. Specifically, it states the circumstances under which computers can initiate connection and disconnection attempts and how their targets should respond in order to establish or cease dialogue. This document does not address the other forms of network communication that may be required such as the NTP or ssh protocols, which already have well documented methods of establishing communication.

# 1 Introduction

The delay line contains many computers that communicate with each other using ethernet links. They exchange information using the messaging protocol described in MRO Delay Line documents “Network Message Protocols and Telemetry/Status/Commands Log File Format” and “dlmsg Library” (both by John Young). However, this information exchange cannot be achieved unless connections between the computers are firstly established in an agreed-upon and orderly fashion.

This document describes how the computers should connect to each other so that messages can be passed between them as required for a functioning delay line. We assume that the reader has a basic knowledge of C programming and the concept of interprocess communication using unix ports and sockets.

# 2 Definitions

For clarity we define the following:

- *Source:* A computer program that generates delay line messages.
- *Sink:* A computer program that receives delay line messages.
- *Command, command status, status, telemetry:* All are message types as set out in the Network Message Protocols document.
- *Controller:* A computer program that sends commands (a “command source”). Normally the workstation is the only controller, but if there is a test program running on another computer that sends commands then it is also a controller.
- *Controllee:* A computer program that accepts commands (a “command sink”). Controllers and controllees need not be running on separate computers.
- *Port1, port2:* Unix ethernet port numbers. Port1≠port2. Port1 is used for status, telemetry and command messages. Port2 is used for command data messages. The values have not yet been defined but the current de-facto standard is to use ports 5000 and 5001.
- *SocketA, SocketB,...,socketF:* Unix sockets, which should be created using David Buscher's socketlib library (distributed with the serialise library described in “Serialise Network Message Protocol” by David Buscher).
- *ProgamA, programB:* Two programs that communicate with each other via a socket connection.

# 3 The Connection Protocol

There are actually two connection protocols. The first is for establishing connections where commands, status and telemetry information is exchanged and the second is for command data, which has simpler requirements. In both cases, the protocol is the same whether the delay line network is being initialised or a connection is being restored due to a prior fault or a deliberate disconnection.

## *3.1 Command, status and telemetry connection protocol*

This protocol makes use of the fact that a given controllee only sends telemetry and status information to one other program (the controller) and only expects commands to arrive from that same program.

1. Controllers should be initialised (possibly using CreateListenPort() from socketlib) so that they listen for connection attempts from anywhere on socketA using port1.
2. Controllees should initiate a connection with a controller by creating socketB (possibly by using ConnectToServer()) which is intended for transmission of status and/or telemetry messages to the controller on port1. Controllers do not initiate such connections[1].
3. When the controller notices that a connection is being attempted on port1, it creates socketC to receive subsequent messages from the controllee on port1 and to send any commands to it (also using port1). As port1 connection attempts arrive from other sources, further sockets are created as necessary.
4. When the controllee notices that socketC exists (that is, its attempt to create socketB was successful), it should start to listen for incoming commands from the controller using socketB.
5. The connection is now established. The controllee can use socketB to send status and telemetry information to and receive commands from the controller. The controller can use socketC to send commands to and receive telemetry and status information from the controllee.

## *3.2 Command data connection protocol*

For command data, a command data sink can expect command data to arrive from anywhere, but it does not need to send anything back to the source.

1. Command data sinks should be initialised so that they listen for command data from anywhere using port2 on socketD.
2. A command data source initiates communication with a command data sink by creating socketE which is intended for transmission of command data using port2.
3. The command data sink reacts by creating socketF for reception of subsequent command data from this source on port2. As messages arrive from other sources, further sockets are created as necessary.
4. The connection is now established. The command data sink can receive command data on socketF. The command data source can transmit command data on socketE.

---

1 Rationale: Under normal circumstances the controller is an always-online workstation, while controllees are added to or removed from the network as needs dictate. It is sensible for controllees to announce their presence to the controller when they appear on the network – otherwise the workstation would have to periodically poll the network to discover which controllees were currently available.

# 4 The Disconnection Protocol

In the operational delay line, it is an error condition for any program to break an established socket connection, even when this is deliberate, and it should be reported as such. Hence a disconnection can be handled in much the same way whether one of the parties deliberately disconnected or there was a system failure such as a network problem.

The disconnection protocol is the same for all kinds of source and sink:

1. ProgramA ceases communication with programB by ceasing to write to and/or read from the corresponding socket. It then destroys the socket itself.
2. ProgramB notices communication with programA has stopped. It might do this by timing out if it was expecting data from programA, or by noticing that messages to programA fail to be sent, or by receiving a “hangup” signal from the socket (this might take several minutes).
3. ProgramB then stops transmitting via the socket in question and destroys that socket. The disconnection process is now complete.