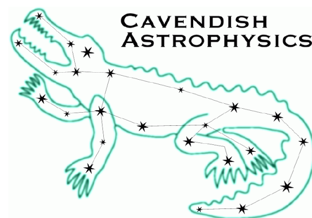
MRO Delay Line

`dlmsg` Library

John Young
jsy1001@cam.ac.uk

rev 0.5

04 September 2009



Cavendish Laboratory
JJ Thomson Avenue
Cambridge CB3 0HE
UK

Table of Contents

1 Introduction....3
1.1 Portability....3
1.2 Use of malloc....4
2 Building and Installing....4
2.1 Prerequisites....4
2.1.1 Install build tools....4
2.1.2 Install GLib....4
2.1.3 Build and Install Exception and Serialise libraries....4
2.2 Build dlmsg Library and Test Programs....4
2.3 Run Test Programs....5
2.4 Install Library....5
3 Header Files to Include....5
4 Generic Functions....6
5 Server/client Architecture....6
6 Status Messages....7
6.1 Sending status....7
6.1.1 Acknowledging commands....8
6.1.2 Adding log/fault messages....8
6.1.3 Advanced Use: Sending concatenated status messages....9
6.2 Receiving status....9
6.2.1 Retrieving command acknowledgements....9
6.2.2 Retrieving log/fault messages....10
6.2.3 Advanced Use: Receiving concatenated status messages....10
7 Telemetry Messages....10
7.1 Sending telemetry....10
7.1.1 Sending concatenated telemetry messages....10
7.2 Receiving telemetry....11
7.2.1 Receiving concatenated telemetry messages....11
8 Command Messages....11
8.1 Sending commands....12
8.2 Receiving commands....12
9 Data Messages....13
9.1 Sending data messages....13
9.2 Receiving data messages....14

Objective

To provide sufficient information to allow developers working on the MRO Delay Lines to make use of the dlmsg library.

Scope

This manual outlines how to call functions in the dlmsg library in order to send and receive status, telemetry, command and command data messages. This document should be read in conjunction with:

- The protocols document “Network Message Protocols and Telemetry/Status/Commands Log File Format”.
- The automatically-generated reference documentation for dlmsg. This documentation is generated by Doxygen from comments embedded within the source code. Refer to the section Build dlmsg Library and Test Programs for instructions on generating a copy of the reference documentation.
- The manual for the Serialise library, which describes how to set up socket connections over the network.
- The manual for the exception-handling library, so you know how to catch exceptions thrown by the messaging code.

1 Introduction

The dlmsg library contains code for sending and receiving all four types of message identified in the protocols document:

- Status messages
- Telemetry messages
- Command messages
- (Command) data messages

Support for concatenated status messages and concatenated telemetry messages (as defined in the protocols document) is included.

It is intended that all sub-systems of the prototype delay line use dlmsg, in order to reduce the overall development time and make messaging more reliable. The library hides details of the message formats from callers, so the formats can straightforwardly be modified if necessary. When compiled without debugging symbols, the size of the dlmsg binary is approximately 40 kilobytes, so there should be no problem using dlmsg on embedded systems.

The library does not provide facilities for socket I/O event-handling or buffering of messages (these are available as part of the workstation software).

1.1 Portability

The dlmsg library has been written in ANSI C (C99), and uses only ANSI C standard library functions, plus GLib 1.2 or 2.x and the Serialise (which requires the sockets library) and Exception libraries.

Hence it should be straightforward to port the library to various platforms. The library has been tested on Linux (Fedora Core 5 with kernel 2.6.15, glibc 2.4, gcc 4.1.0; Fedora Core 6 with kernel 2.6.22, glibc 2.5, gcc 4.1.2; and Fedora Core 10 with kernel 2.6.27, glibc 2.9, gcc 4.3.2) and QNX Neutrino (versions 6.2.1 and 6.4.0).

1.2 Use of malloc

The dlmsg library requires GLib. GLib objects and functions are only used internally; your code need not invoke GLib explicitly.

The dlmsg library is written in an object-oriented style, as outlined in the coding standards document "Coding and Documentation Guidelines for Prototype Delay Line Software".

Both of these design decisions have led to the messaging library making extensive use of dynamic memory allocation. In particular, the library calls g_malloc() and g_free() (as well as GLib functions that themselves call g_malloc() and g_free()). By default g_malloc() calls the system (i.e. C standard library) malloc, but this behaviour can be changed if necessary — refer to the GLib documentation for more details.

2 Building and Installing

2.1 Prerequisites

2.1.1 Install build tools

You will need gcc and Gnu make to build the library, and Doxygen to generate the reference documentation.

2.1.2 Install GLib

The messaging library requires GLib version 2.x (you can use version 1.2 if necessary, but this has not been tested as thoroughly; QNX Neutrino only comes with 1.2). It has been fully tested with versions 2.10, 2.12 and 2.18.

Packages for GLib are available for most Linux distributions. You will need to install the "development" package (this is called glib2-devel on Fedora Core).

2.1.3 Build and Install Exception and Serialise libraries

Copy the Exception library source to your desired build host.

> To obtain the source from our subversion repository, use the following command:
>
> shell> `svn checkout svn://apcsthatt/dl/libs/exception`
>
> If you don't have a subversion client on the build host you can run `svn checkout` on apcsthatt then copy the source files to the build host.

If building for Linux, first check that your shell on the build host exports the OSTYPE environment variable. If not you can specify it on the the command line when invoking make, e.g.: `OSTYPE=linux make`.

On the build host, build and install (by default, to subdirectories of `/usr/local/`) by typing `make`, becoming root, then typing `make install`.

The procedure for building and installing Serialise is identical, except that the repository URL you should supply to **svn checkout** is svn://apcsthatt/dl/libs/serialise.

2.2 Build dlmsg Library and Test Programs

Retrieve the source for dlmsg and copy it to your desired build host. The subversion repository URL for dlmsg is svn://apcsthatt/dl/libs/dlmsg.

Type `make` to build the library and test programs.

Build the reference documentation by typing `doxygen Doxyfile`. This will generate HTML-format documentation in a subdirectory `html`, plus LaTeX source in a subdirectory `latex`. The latter can be used to generate documentation in PDF format: change to the `latex` subdirectory and type `make` to generate a file `refman.pdf`.

2.3 Run Test Programs

There are four pairs of test programs. Each pair communicate using the loopback interface, and so should be run on the same computer (use two terminal windows).

- **simpleclient** and **simpleserver**: send/receive status messages
- **test_sendtele** and **test_recvtele**: send/receive telemetry messages
- **test_sendcmd** and **test_recvcmd**: send/receive command messages
- **test_senddata** and **test_recvdata**: send/receive command data messages

It should be obvious from the terminal output whether a given pair of programs are communicating successfully.

2.4 Install Library

Install the (static) library and C header files under `/usr/local/` by becoming root and typing `make install`.

3 Header Files to Include

Code that calls dlmsg functions must `#include` the appropriate header file(s). These are as follows:

- `status.h`: Definitions related to status messages
- `telemetry.h`: Definitions related to telemetry messages
- `cmd.h`: Definitions related to command messages
- `data.h`: Definitions related to (command) data messages

Each of these four header files has a comment at the top of the file listing the headers to include before including the file itself. For example, `status.h` begins with:

```
/** @addtogroup dlmsg */
/*@{*/

/**
 * @file status.h
 * Definitions related to status messages.
 *
 * @author John Young
 *
 * Code that includes this file should first include:
 * - <glib.h>
 * - "exception.h"
 * - "serialise.h"
 * - "datatypes.h"
 * - "message.h"
 * - "vclient.h"
 * - "ack.h"
 */
```

Note that an alternative approach would have been for each of the top-level header files to contain the appropriate `#include` statements.

4 Generic Functions

The functions whose prototypes are in `message.h` are applicable to all types of message. The most useful one is probably get_msg_type(), which returns an enum giving the message type — this is intended for use by a program that receives several types of messages from the same socket (such as the telemetry/status server described below). The deserialiser passed to get_msg_type() is reset back to the start of the message on exit, so it can then be passed to a message-type-specific decoding function.

The other generic functions are really only useful within dlmsg:

- get_message_type_noreset(): as get_msg_type() but doesn't reset deserialiser
- skip_tuple(): skip arbitrary tuple in serialised message
- array_typecode(): return serialise typecode for an array with the specified SerialType

5 Server/client Architecture

The telemetry and status protocols defined in the protcols document are based on the concept of a server (the delay line Workstation) that listens on a pre-arranged TCP port, to which clients (e.g. trolley micros, VME system) connect. Telemetry and status messages received from each client are handled by the server, until the client breaks the connection.

The messaging library allows for the possibility that a single computer may act as several "virtual clients", each sending a distinct set of telemetry and/or status. A program that sends telemetry and/or status (and isn't just forwarding messages verbatim) must create an object to store the state of each virtual client that it implements:

```
VClient *pVClient;
ExcStatus *status;

/* Initialise error system */
status = ExcNew();

/* Initialise sender state */
pVClient = CreateVClient("VME", status);
```

The pointer to the resulting VClient should be passed to the telemetry/status object constructors CreateTele() and CreateStat(). When it has finished sending messages, the sending program can free the storage associated with the VClient instance by calling the appropriate destructor:

```
DestroyVClient(pVClient, status);
```

Use of the VClient "object" ensures that:

- All messages from the same virtual client contain the same client identifier string and consistent configIds
- The configId is incremented automatically whenever the set of telemetry streams/status items has changed (detected by noting calls to CreateStat() and CreateTele()[1]) since the previous message was sent

---

1This change detection mechanism will be fooled by clients that create all of their status/telemetry objects during initialisation, then later switch between transmitting different subsets of them. If your client does this, it should call VClientSetConfiguring() before each switch.

Note that the messaging protocol and VClient implementation allow each virtual client to divide its telemetry streams/status items amongst more than one sequence of telemetry/status messages. Each message can be concatenated or unconcatenated. For example, a client could send two telemetry messages every 0.1s, one containing streams sampled at 5kHz, and one containing streams sampled at 10Hz.

However, a more complex server implementation is required to cope with virtual clients that send several sequences. To facilitate such an implementation, the messaging library provides functions to efficiently identify which sequence a message belongs to: get_XXX_seq_key() and get_XXX_seq_hash().

6 Status Messages

These are the most complicated messages, as they can include both boolean and numerical values (with associated labels), as well as command acknowledgements and log/fault notifications.

However, all delay-line sub-systems will need to send status messages to the Workstation, so we might as well treat status messages first!

The same “class”, StatUnit, is employed for both sending and receiving messages. A StatUnit instance represents the content of an unconcatenated status message (a list of command acknowledgements, a list of logs and faults, plus a set of boolean and numeric status items with a common timestamp).

6.1 Sending status

After reading this section, please examine the test program source code in `simpleclient.c`, which combines the function calls outlined here in a complete program. Some simple error checking is also included.

Create a StatUnit instance (status object) to store the current status of the sub-system, using CreateStat(). The following example code creates a status object with two boolean status items and two numerical status items. Labels are associated with the status items, but their values are not initialised.

```
#include <glib.h>
#include "exception.h"
#include "serialise.h"
#include "datatypes.h"
#include "message.h"
#include "vclient.h"
#include "ack.h"
#include "status.h"

VClient *pVClient;
StatUnit *myStat;
ExcStatus *status;
char *boolLabel[] = {"SteeringOn", "Tracking"};
char *numLabel[] = {"Jitter1", "Error1"};
char *numUnits[] = {"nm", "nm"};

/* Initialise error system */
status = ExcNew();

/* Initialise sender state */
pVClient = CreateVClient("myClient", status);

/* Create status object */
myStat = CreateStat(pVClient, sizeof(boolLabel)/sizeof(char *), boolLabel,
                    sizeof(numLabel)/sizeof(char *), numLabel, numUnits,
```

```
                  status);
```

Note that code examples in later sections will omit the necessary #include statements, for the sake of brevity.

The sub-system should open a socket connection to the Workstation. Please refer to the Serialise manual for more details.

```
  socket = ConnectToServer("apcsthatt", port, status);
```

The boolean status values stored in the StatUnit instance would normally be updated whenever the sub-system status changes, by calling StatSetBool():

```
  StatSetBool(myStat, "SteeringOn", TRUE, status);
```

The numerical status values can be updated just prior to transmission, by calling StatSetNum(). The timestamp for the status values should also be set immediately before transmission:

```
  #include <sys/time.h>
...
  struct timeval tv;
  struct timezone tz;
...
  StatSetNum(myStat, "Error1", val, status);
  gettimeofday(&tv, &tz);
  StatSetUtc(myStat, (tv.tv_sec + 1e-6*tv.tv_usec), status);
```

Send a status message by calling StatSendMsg():

```
  StatSendMsg(myStat, socket, status);
```

When the sub-system has finished sending messages, it should close the socket connection. The storage associated with the StatUnit instance can be freed by calling DestroyStat():

```
  #include <unistd.h>
...
  close(socket);
  DestroyStat(myStat, status);
```

6.1.1 Acknowledging commands

To add a new command acknowledgement to the status object (you should do this whenever a command is parsed), call StatAckFromArgs():

```
  char *cmdSource;
  Int32 cmdTag;
  Bool parseFlags[3];
...
  StatAckFromArgs(myStat, cmdSource, cmdTag, parseFlags, status);
```

The list of acknowledgements is automatically transmitted each time StatSendMsg() is called. After sending the message, you should clear the acknowledgement list with:

```
  StatClearAck(myStat, status);
```

6.1.2 Adding log/fault messages

To pass on a log message or report a fault which has occurred within the sub-system, call StatAddLog(). This may be called any number of times to store multiple logs and/or faults. The logs and faults will then get included in the next status transmission.

The sub-system should only report each log/fault once, so call StatClearLog() after

transmitting the status message.

6.1.3 Advanced Use: Sending concatenated status messages

Only send concatenated messages if you need separate timestamps for different status items. For each subset with a common timestamp, create a StatUnit instance using CreateStat(). To send a concatenated message, pass pointers to all of the StatUnit instances to stat_send_concat() (which takes a variable number of arguments):

```
  int socket, nStat;
  ExcStatus *status;
  StatUnit *pStat1, *pStat2, ...;
...
  stat_send_concat(socket, status, nStat, pStat1, pStat2, ...);
```

An alternative interface is provided for clients that don't know at compile-time how many status units they will be concatenating:

```
  StatUnit *ppStat[];
...
  stat_send_concat_alt(socket, ppStat, nStat, status);
```

In the last example, ppStat is an array of pointers to the StatUnit instances.

6.2 Receiving status

After reading this section, please examine the test program source code in `simpleserver.c`, which combines the function calls outlined here in a complete program. Some simple error checking is also included.

Receiving status is dealt with in less detail, since only the Workstation needs to do it. The status items sent by a particular client are not known until a message is received, so a StatUnit instance is created from the content of the first status message. Once a socket connection has been established, the message is read into a Deserialiser using ReadMessage() (refer to the Serialise manual). A pointer to the deserialiser is passed to StatFromMsg() to create the status object:

```
  StatUnit *pStat;
  Deserialiser *message;
  ExcStatus *status;
  int socket;
...
  message = ReadMessage(socket, status);
  pStat = StatFromMsg(message, status);
```

Status values can now be retrieved using StatGetBool() and StatGetNum(). The status items present can be queried using StatBoolLabels() and StatNumLabels().

Subsequent status messages in the same sequence (you can check whether they are using get_statmsg_seq_key() or get_statmsg_seq_hash()) should contain an unchanged set of status items. The UpdateStat() method is used to update the values in a status object from a serialised message — it will throw an exception if the set of status items has changed since the call to StatFromMsg().

Destroy the deserialiser and status object when you are finished with them:

```
  DestroyDeserialiser(message, status);
  close(socket);
  DestroyStat(message, status);
```

6.2.1 Retrieving command acknowledgements

Once a message has been decoded using StatFromMsg() or UpdateStat(), you can check whether it contained the acknowledgement to a particular command by passing the command tag to StatGetAck(). If there is a matching acknowledgement, this will return a pointer to an Ack instance. Use the AckGetXXX() macros to access the various components of the acknowledgement.

6.2.2 Retrieving log/fault messages

Once a message has been decoded, call StatGetLogList() to retrieve a linked list (as a GSList — see GLib documentation) of structs, each containing a log/fault message and its associated metadata.

6.2.3 Advanced Use: Receiving concatenated status messages

Use stat_from_concat() to create a set of StatUnit's from a serialised message. Use stat_update_concat() these from subsequent messages in the same sequence. Note that these functions also work with unconcatenated messages.

7 Telemetry Messages

The code to send and receive telemetry messages has a very similar structure to that for status messages. However, not all StatUnit methods have telemetry equivalents, as telemetry messages have fewer distinct components.

A single “class”, TeleStream, is employed for both sending and receiving messages. A TeleStream instance represents the content of an unconcatenated telemetry message, i.e. the properties of the telemetry stream (name, sample rate, chunk length etc.), plus a single chunk (typically 0.1–1 second's worth) of telemetry data.

7.1 Sending telemetry

After reading this section, please examine the test program source code in `test_sendtele.c`, which combines the function calls outlined here in a complete program.

Initialise the telemetry stream using CreateTele():

```
  VClient *pVClient;
  TeleStream *myStream;
  ExcStatus *status;
...
  myStream = CreateTele(pVClient, 1, 0, "testStream",
                  SAMPLES_PER_SEC, SAMPLES_PER_CHUNK,
                  SER_FLOAT32, "nm", status);
```

Send each chunk of telemetry by calling SendTele(), which takes the chunk value array and timestamp as arguments:

```
  int socket;
  Float32 *chunk;
  Float64 utc;
...
  TeleSendMsg(myStream, socket, chunk, utc, status);
```

7.1.1 Sending concatenated telemetry messages

This is a more efficient way of transmitting multiple streams (the Workstation will receive fewer messages and hence fewer “interrupts”).

Create a TeleStream instance for each stream using CreateTele(). To transmit a concatenated message, first assign the next chunk of data to each object using TeleSetChunk(), then pass all of the objects to tele_send_concat(). tele_send_concat_alt() provides an alternate interface, analogous to that of stat_send_concat_alt().

7.2 Receiving telemetry

After reading this section, please examine the test program source code in `test_recvtele.c`, which combines the function calls outlined here in a complete program.

A TeleStream instance is created from the content of the first telemetry message. Once a socket connection has been established, the message is read into a Deserialiser using ReadMessage(). A pointer to the deserialiser is passed to TeleFromMsg() to create the telemetry object:

```
  TeleStream *pStream;
  Deserialiser *message;
  ExcStatus *status;
...
  message = ReadMessage(socket, status);
  pStream = TeleFromMsg(message, status);
```

The first data chunk can now be accessed using TeleGetChunk():

```
  Float64 utc;
  void *chunk;
...
  chunk = TeleGetChunk(pStream, &utc, status);
```

Subsequent messages in the same sequence (you can check this using get_telemsg_seq_key() or get_telemsg_seq_hash()) are decoded using TeleRecvMsg(), which for convenience returns the new data chunk and timestamp (calling TeleGetChunk() afterwards also works):

```
  message = ReadMessage(socket, status);
  chunk = TeleRecvMsg(pStream, message, &utc, status);
```

The library keeps track of the position in the stream — this can be queried with TeleGetSampleIndex(). If the received messages do not make up a contiguous data stream, an exception is thrown.

Note that the receiving code should not assume the data type of a stream. The data type can be queried using TeleGetType() once the first message has been decoded. Other stream properties can be queried using the TeleGetXXX() macros. These properties (sampling rate, chunk length etc.) should not change once the stream has been started.

7.2.1 Receiving concatenated telemetry messages

Use tele_from_concat() to create a set of TeleStream's from a serialised message. Use tele_update_concat() to update the objects from subsequent messages. After decoding a message, use TeleGetChunk() to retrieve the new data from each TeleStream instance in the array.

Note that these functions also work with unconcatenated messages.

8 Command Messages

The Cmd “class” is employed for sending and receiving command messages. Both

Cmd and Data (for command data messages) inherit from an abstract parent “class” CmdData (because the message formats are identical).

8.1 Sending commands

After reading this section, please examine the test program source code in `test_sendcmd.c`, which combines the function calls outlined here in a complete program.

Create a Cmd instance with CreateCmd(), then initialise its immutable attributes (those that don't change from one transmitted message to the next) by calling InitCmd(). In the case of commands, the only immutable attribute is the name of system sending the commands — contrast this with the data messages described below.

```
  Cmd *pCmd;
...
  pCmd = CreateCmd(status);
  InitCmd(pCmd, "mySys", status);
```

Two alternative methods for sending commands using the initialised Cmd instance are provided. SendCmd() is passed an array of command parameters:

```
  Int16 paramVal[] = {1, 11, 21};
...
  SendCmd(pCmd, socket, "DoNothing", SER_INT16, 3, paramVal, status);
```

SendCmdAlt() is passed the parameter values at the end of the argument list:

```
  SendCmdAlt(pCmd, socket, "DoNothing", SER_INT16, 3, status, 1, 11, 21);
```

To send a command that takes no parameters, pass SER_NONE for the parameter type, zero for the number of parameters, and (if applicable) NULL for the parameter array:

```
  SendCmdAlt(pCmd, socket, "NoParam", SER_NONE, 0, status);
  SendCmd(pCmd, socket, "NoParam", SER_NONE, 0, NULL, status);
```

Use the same Cmd instance for sending commands to all destinations. This has the advantage that each command will have a unique tag.

The storage used by the Cmd instance is freed by calling DestroyCmd().

8.2 Receiving commands

After reading this section, please examine the test program source code in `test_recvcmd.c`, which combines the function calls outlined here in a complete program.

To receive commands, first create a Cmd instance using CreateCmd(). Serialised command messages are then decoded by passing this to RecvCmd(). The command string can then be accessed with CmdGetCmd() and the associated parameters with CmdGetParam() or CmdGetParamVal():

```
  Deserialiser *message;
  int socket;
  ExcStatus *status;

  Cmd *pCmd;
  SerialType paramType;
  int len;
  char *cmd;
...
```

```
  pCmd = CreateCmd(status);
...
    message = ReadMessage(socket, status);
    RecvCmd(pCmd, message, status);

    /* Display contents */
    printf("Received command from %s:\n", CmdGetSourceId(pCmd, status));
    printf("  Tag: %d\n", CmdGetTag(pCmd, status));
    cmd = CmdGetCmd(pCmd, status);
    printf("  Cmd: %s\n", cmd);
    if(strcmp(cmd, "DoNothing") == 0)
    {
      /* Verify type/length of parameter array */
      (void) CmdGetParam(pCmd, &paramType, &len, status);
      assert(paramType == SER_INT16);
      assert(len == 3);
      /* Print parameters */
      /* Note: could use return from CmdGetParam instead of CmdGetParamVal */
      printf("  Param: %d %d %d\n",
           CmdGetParamVal(pCmd, 0, Int16, status),
           CmdGetParamVal(pCmd, 1, Int16, status),
           CmdGetParamVal(pCmd, 2, Int16, status));
    }
```

The CmdGetParamVal() macro must be passed the C data type of the parameters, so be careful to use the appropriate type, based on the decoded command string.

Note that the first call to RecvCmd() sets the immutable attributes of the command object. Hence if you try to use the same object to decode commands from different sources, an exception will be thrown. This will not be an issue in practice, as all commands are sent by the Workstation.

9 Data Messages

The Data “class” is employed for sending and receiving command data (henceforth just “data”) messages. Both Data and Cmd (see above) inherit from an abstract parent “class” CmdData.

9.1 Sending data messages

After reading this section, please examine the test program source code in `test_senddata.c`, which combines the function calls outlined here in a complete program.

Create a Data instance for each data “stream” (messages with the same data label and destination) using CreateData(). Initialise the immutable attributes (those that don't change from one transmitted message to the next) of each object by calling InitData():

```
  Data *pData;
  SerialType dataType = SER_INT32;
  int dataLen = 3;
...
  pData = CreateData(status);
  InitData(pData, "mySys", label, dataType, dataLen, status);
```

Two alternative methods for sending data messages using the initialised Data instance are provided. SendData() is passed an array of data:

```
  Int32 dataVal[] = {1, 11, 21};
```

```
...
  SendData(pCmd, socket, dataVal, status);
```

SendDataAlt() is passed the parameter values at the end of the argument list:

```
 SendDataAlt(pCmd, socket, status, 1, 11, 21);
```

The storage used by the Data instance is freed by calling DestroyData().

9.2 Receiving data messages

After reading this section, please examine the test program source code in `test_recvdata.c`, which combines the function calls outlined here in a complete program.

To receive data messages for a particular "stream", first create a Data instance using CreateData(). Serialised messages are then decoded by passing this to RecvData(). The data label can then be accessed with DataGetLabel() and the associated data values with DataGetData() or DataGetVal():

```
  Deserialiser *message;
  int socket;
  ExcStatus *status;

  Data *pData;
  SerialType dataType;
  int len;
  void *data;
  Int32 *dataInt32;
  char *label;
...
  pData = CreateData(status);
...
    message = ReadMessage(socket, status);
    RecvData(pData, message, status);

    /* Display contents */
    printf("Received data from %s:\n", DataGetSourceId(pData, status));
    printf("  Tag: %d\n", DataGetTag(pData, status));
    label = DataGetLabel(pData, status);
    data = DataGetData(pData, &dataType, &len, status);
    printf("  Label: %s\n", label);
    if(strcmp(label, "DoNothing") == 0)
    {
      assert(dataType == SER_INT32);
      assert(len == 3);
      /* Note: could use DataGetVal() instead of following code */
      dataInt32 = (Int32 *) data;
      printf("  Data: %d %d %d\n",
           dataInt32[0], dataInt32[1], dataInt32[2]);
```

DataGetVal() must be passed the correct C data type, so be careful to use the appropriate type, based on the decoded label.

Note that the first call to RecvData() sets the immutable attributes of the data object. Hence if you try to use the same object to decode messages with e.g. different labels, an exception will be thrown.