MarkerBroker: collect 3/6 DOF marker data: user's manual

Johnston, Daniel F.

Publisher's version / Version de l'éditeur:
https://doi.org/10.4224/21272482
IMTI ITFI Publication; no. IMTI-TR-021, 2003-09-05

INTEGRATED MANUFACTURING TECHNOLOGIES INSTITUTE

INTSTITUT DES TECHNOLOGIES DE FABRICATION INTÉGRÉE

Pages: 70

REPORT
RAPPORT

Date: 2003 September 05

Fig.
Diag. 13

SAP Project #
de Projet de SAP 55-W99

For
Pour IMTI

Unclassified
(Classification)

IMTI-TR-021 (2003/09)

MarkerBroker – Collect 3/6 DOF marker data
User's Manual

Submitted by
Présenté par Daniel F. Johnston

First Author
Auteur Premier Daniel F. Johnston

Approved
Approuvé Dr. Gian Vascotto
Director

Abstract

A software program has been created to simplify the collection of data from position tracking hardware and to simplify the creation of applications that use this data. Standard configuration files allow the program to adapt to different configurations and requirements. The document describes how to build and use the program, and also some design information for those who wish to make improvements.

Contents

1 Introduction to documentation **1**

1.1 Main chapters 1

1.2 Appendixes 2

2 Introduction to MarkerBroker Software **5**

3 Introduction to 6 DOF position markers **7**

3.1 Position measurements with markers 7

3.2 Data communication types 10

3.3 The lack of data standards 11

4 Description of MarkerBroker and its features **15**

4.1 Computers that support MarkerBroker requirements 15

4.2 Important features of MarkerBroker 16

4.3 How to use the MarkerBroker application 17

4.4 Format of the sent marker data message 23

4.5 What might go wrong and why 23

5 Configuration File **25**

5.1 General features of configuration files 27

5.2 Header section 27

5.3 Description and Author sections 28

5.4 Setting section 29

5.5 Network connections 31

5.6 Serial line connections 32
5.7 The tail of the configuration file 33

6 MarkerBroker install and build **35**
6.1 Computer systems which support MarkerBroker 35
6.2 How to obtain the source for MarkerBroker 36
6.3 Project directory structure 36
6.4 General comments on the building process 40
6.5 Building the MarkerBroker applications 42
6.6 Building the project documentation 43
6.7 Additional build options 43

7 MarkerBroker design considerations **45**
7.1 Requirements for a Marker Broker 45
7.2 General operation of MarkerBroker 46
7.3 Specific to the main application 47
7.4 Specific to connection applications 47
7.5 Advantages of using MarkerBroker 48
7.6 Disadvantages to using MarkerBroker 48
7.7 Future directions for MarkerBroker 48

A MarkerBroker - DTD and configuration files **49**

B MarkerBroker - Connection applications source code **59**

List of Figures

2.1 Where to use MarkerBroker 5

3.1 NDI Polaris optical position sensor 8
3.2 Ascension system for magnetic tracking 9
3.3 Wireless magnetic tracking backpack 10
3.4 Network communication for passive optical tracking 11
3.5 Some hardware supported by TrackD 12

4.1 Main components of MarkerBroker 16
4.2 Main window of MarkerBroker 18
4.3 Default marker window 21
4.4 Current configuration window 22

5.1 Configuration file structure 26

6.1 Directory structure of MarkerBroker 37
6.2 Main classes defined for MarkerBroker 42

Chapter 1

Introduction to documentation

The documentation for the `MarkerBroker` software application is provided as a set of chapters in this document. Just read the 'Summary' chapter if you want to find out what the software can do. Remaining chapters describe the background to, applications for, and the features of this software utility - and how to use them.

1.1 Main chapters

Chapters:

1. **Introduction to documentation:**

   An introduction to the documentation for the `MarkerBroker` application program. The chapter you are now reading.

2. **Introduction to MarkerBroker Software**:

   A brief description of `MarkerBroker`: Why was it created? What can the software do? Where can it be used? The summary would be useful to anyone who needs a brief description of `MarkerBroker` features and potential areas of application.

3. **Introduction to 6 DOF Sensors and Markers:**

   A general introduction to position and orientation sensors will be found in this chapter. The emphasis will be on the different types of sensor, the types of communication, and the (lack of) standards. Read this chapter if you are not familiar with position tracking devices.

4. **Description of MarkerBroker Application:**

A detailed guide to the `MarkerBroker` application. The chapter describes the function of every control and menu option. Users will learn how the application should work, and what are some probable reasons if it does not. Anyone who would like to use the `MarkerBroker` application software should read this chapter.

5. **Marker Configuration Files using XML:**

   A description of what is XML and how is it used to describe position sensors and their connections. All users of the application will eventually need to examine this chapter, and the first appendix, so they can connect the application to their tracking hardware.

6. **Install and Build:**

   A chapter to describe what features are needed for a computer to run the `MarkerBroker` application, how to install the software source code, and how to build the application software from this source. This chapter is required reading for the first time user, but should not be needed again until the application is modified.

7. **Design Considerations:**

   This chapter is provided for those who are trying to understand how the software works, or understand the source code. It is intended that this chapter be a record of the design ideas that were used to create the application, and is not intended for someone who intends to only be a user. There are also a few comments about possible future directions for `MarkerBroker`.

1.2 Appendixes

Additional information is provided in the two appendixes;

A **Configuration: Several detailed configuration files**

This appendix provides listings of the required Document Type Definition (DTD) file that is referenced in every configuration file but not normally seen by the end user. Then the appendix includes a few listings of complete configuration files for real hardware.

B **Supported: Source code for supported hardware**

Connection software has been written for the position tracking hardware that we use. The source code for these programs is reproduced here and described. These programs can serve as examples for the development of support for future hardware.

All `MarkerBroker` source code files at the time this report was generated are reproduced in the CD that can be obtained. This computer disk will also

contain all the files described in this document, all the information needed to create the application and the documents, and a copy of the documentation in several different forms.

Chapter 2

Introduction to MarkerBroker Software

This chapter describes, in brief, the features and operation of the `MarkerBroker` software application. It will explain why the program was created, what can it do, and how it is intended to be used.

Position markers are attached to objects, machines, and humans so that the 3D location (and orientation) of these objects can be monitored. Data about position will be used by computer programs to implement many desirable applications. It is difficult to write applications that support the many different marker types and vendor specific data formats.

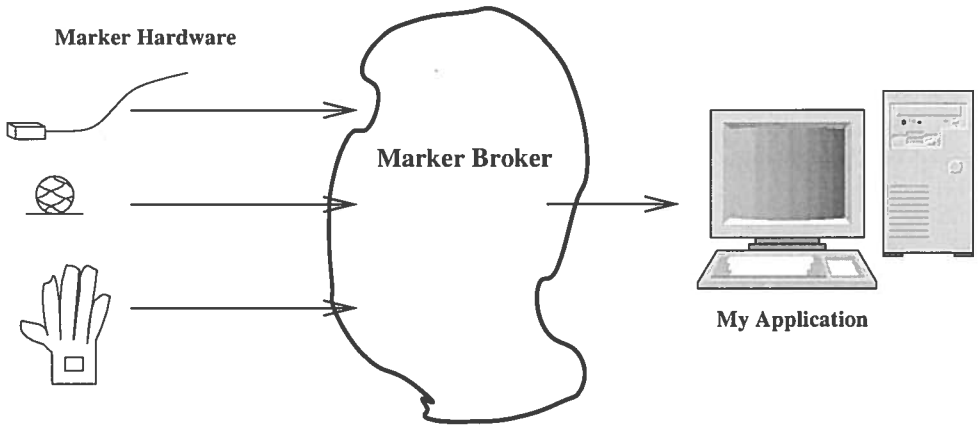

Figure 2.1: Where to use MarkerBroker

`MarkerBroker`, as shown in the figure, acts as an intermediary between the

hardware and the end-user application. As a 'broker' of data formats, it will gather all the data and create a single, simple data set that can easily be used for any marker positioning purpose. This simplification supports the rapid development of new applications.

It is expected that any 3D position marker broker would have to adapt easily to any combination of known hardware. Existing, common, 3D position tracking systems may have to coexist with new and prototype sensors. The `MarkerBroker` application has been designed to use a configuration file, written in a standard (XML) format. This file will automatically modify the behavior of the `MarkerBroker` to satisfy your requirements.

Each position sensor is connected to `MarkerBroker` with a very small and easy-to-modify sub program. Several existing sub programs (called "connection applications") are provided for common tracking hardware. Clearly identified requirements for these connection applications makes it easy to create new ones that support new devices and unique requirements.

So the `MarkerBroker` software was designed to provide three key features:

- Simplified output for rapid development of applications.
- Behavior modified using XML files to match end-user requirements.
- Clear and simple requirements for connection applications, which will supports rapid development of software to support new/different sensor hardware.

Chapter 3

Introduction to 6 DOF position markers

This chapter gives some background on why it is desirable to measure the position and orientation of things. It is necessary to record the 3D position (x, y, z) and complete orientation angles (e.g. yaw, pitch, roll). A complete position description requires six data values, i.e. six Degrees Of Freedom (DOF).

This chapter then describes two very common types of position tracking hardware - based on the class of communication used. To finish, the chapter explains the problem with hardware control and data transfer.

3.1 Position measurements with markers

It is frequently necessary to measure the position of objects as they move. This might be for reasons of security, e.g. intruder detection and tracking via video cameras. It might be for navigation, where ships and airplanes use radar and microwave to plan and verify their voyages, and to avoid other vessels. Position tracking can improve the efficiency of companies by monitoring in real time the location of a truck and its cargo anywhere in a country, or pinpointing the current location of a taxi in a city.

Industrial, medical and scientific problems also can have a requirement to track the position and orientation of parts and tools, of machines as they move, of people and their body position. This tracking could be for resons of safety, to measure the energy use or stress applied to the measured body, or to provide complex control of other machines. Concrete evidence of the need for tracking can be found in the very large number of hardware systems that are now available to obtain this type of position data. You can select from wired to wireless, magnetic to optical to encoder, active to passive,

single to multi channel.

Encoder systems connect to sensors that are mounted directly on the equipment. The most common sensor provides only joint angle data. Data from these sensors is accurate and immediate. Position information can be derived from these accurate joint angles when the size of the components is known. Modern encoder systems for human use can be provided with counterweights to balance the mass of the position measuring hardware.

Rather than modify existing equipment, and people, to accommodate the mounting of encoders, position and orientation is often measured by attaching a sensor or 'marker' to the object and reading the position of this (these) marker(s). Optical system can use wired or wireless markers whose 3D positions are triangulated by cameras. Multiple sensors are usually required to obtain data about the object's orientation.

Wired, or 'active' markers are usually more accurate, but you must provide for the wires to each marker and their connection back to the measurement system. Wireless, or 'passive', markers are simpler to install and the object can move with little restrictions. All optical systems require a 'line of sight' between marker(s) and cameras. If the view of a marker is blocked, then the system cannot measure that object's position. Using more cameras and extra, redundant, markers can reduce this problem, but optical systems cannot be used where the position marker will usually be hidden from view.

Figure 3.1: NDI Polaris optical position sensor

To sense marker position 'through the body', it is common to use a magnetic tracking system. A single magnetic pulse from one source can be measured by many sensors. Measurement accuracy is affected by the presence of any ferrous material near or between the sensor marker and the magnetic pulse antenna. Measurement does not depend on the markers visibility.

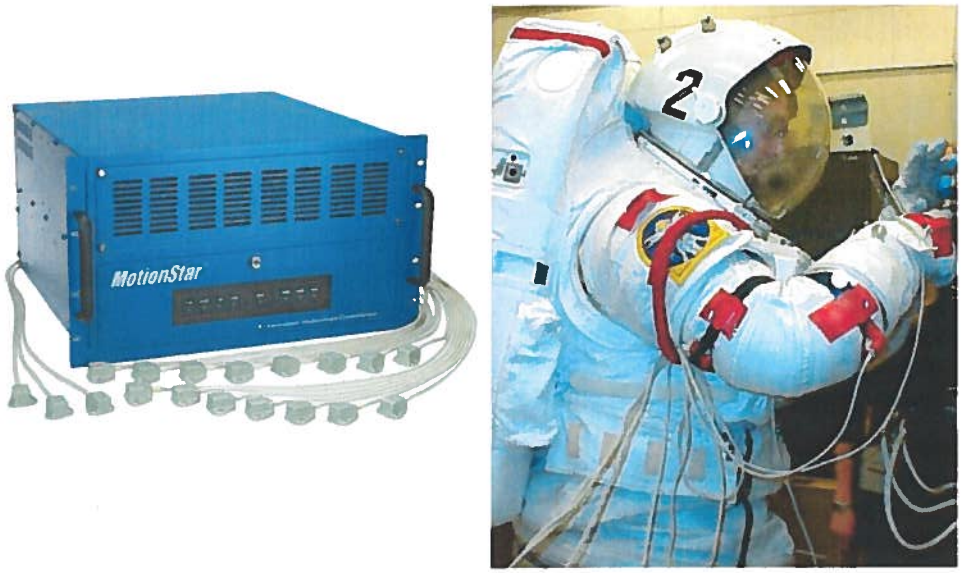

Figure 3.2: Ascension system for magnetic tracking

Each magnetic sensor within range will provide information about its 3D position and orientation. Every sensor will require wires and connected electronics to calculate position from the magnetic measurement. Some vendors provide a 'backpack' to which every sensor is connected. The next figure clearly shows the sensors, the wires, and the pack which contains the electronics. A single wire or wireless connection will then send the measurement data to a base station, and then on to computers.

Figure 3.3: Wireless magnetic tracking backpack

3.2 Data communication types

Just as there is variety in the technology used to obtain position data, there is a wide variety in how the data is formatted for transmission to the end-use application, in how fast the position data values are updated, and with what technology the data is sent/read.

3.2.1 Serial communication

Older tracking systems, and those where the sensor data is updated at a slower rate, often use a serial communication line to transfer data This means that data is transferred one character or digit at a time at speeds, for example, of 960 or 1920 characters per second. This speed is sufficient for a few position markers that change at moderate rates. It is also adequate for many applications where each sensor can provide both position and orientation, i.e. there are fewer markers needed.

The magnetic tracking system shown in figure 3.2 above, and the older but still popular version of this product, use a serial connection as their standard way to communicate data and control. Many applications have been developed that include built-in support for a serial connection to these tracking systems.

3.2.2 Network communication

If the number of tracked positions is large, or if the message containing the position information is verbose, then using character by character data transmission to update positions in the final application will be slow. For these situations vendors now provide tracking systems which use network connections to transfer data at rates of at least 50,000 characters per second.

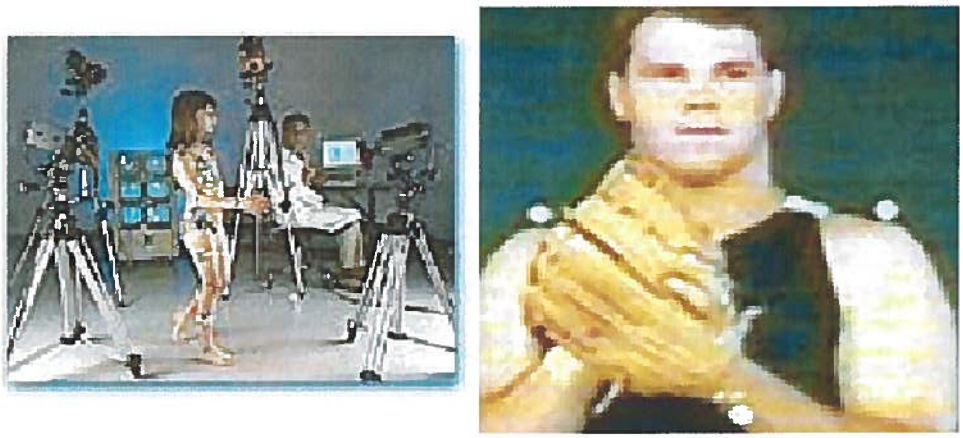

Figure 3.4: Network communication for passive optical tracking

Most modern position tracking systems will support a network connection option for data transfer. Network hardware, once expensive, is now so low in cost that most computers include a network connection as part of the base configuration. This means that network data transfer can be supported even for applications where the data transfer rates do not require it. For example, the magnetic tracking system and backpack version shown in the figures above are also available in a version with a network communication connection.

3.3 The lack of data standards

There is still the problem of understanding this position data. Every vendor has a proprietary data format, or at least a unique way of packaging a common data format. Every vendor has defined a set of communication codes to setup his device and start data transfer. These vendors would prefer that you modify their example code to read position data from their markers, and not use other marker types because you would have to write new software to support them. This is not always possible.

Is there software that can read data from a number of position tracking systems? Yes, there is. Virtual Environments, which often make use of tracked position data, can be created with libraries that have support for 'several' common tracking systems. If you are creating a virtual environment and using only the supported hardware, then you could use such a library for your development. What about new tracking hardware? Well, then you would be back to creating you own software, but now it must be compatible with a complex library that you didn't create and may not understand. Examples: `DIVERSE` from Virginia Tech., `VRJuggler` from Iowa State, and `OpenMask` from IRISA in France.

Figure 3.5: Some hardware supported by TrackD

What about something that is more general? The most well known is the `TrackD` software licensed from VRCO. It does support a large number of common tracking devices, some of which are shown in the figure above, but it does not support them all. Since the software is driven by market acceptance and needs, it does not communicate with unique and newly developed tracking systems. There is some recent support for adding new devices with an optional `TrackD` software development package. `TrackD` is widely used in the Virtual Environment industry - perhaps because

there wasn't anything else to replace it. License fees are not expensive if you are already maintaining a multi-million dollar VE display system but would seem relatively expensive to anyone else.

How else can you create applications which use a variety of position tracking systems without including expensive and/or complex software? How can you reduce the time, cost and difficulty of adding a new position tracking system to your application? How do you support all those different communication techniques and speed? One solution is to use software that can communicate with each device on its own terms, collect all the data into a common and simplified format, and retransmit this data to your application at your desired data rate. `MarkerBroker` is a software application which was designed do this. It is described in the next chapter of this document.

Chapter 4

Description of MarkerBroker and its features

This chapter describes the features and operation of the `MarkerBroker` software application. It will explain what are the program's requirements, what can it do, and how it is intended to be used. The description will include the optional windows provided by the application.

4.1 Computers that support MarkerBroker requirements

A few of the application's features are defined by a combination of design decisions and system requirements.

One of the main design features of the `MarkerBroker` application is that it will create a number of additional sub-processes, each of which is responsible for communication to one source of position markers. Multitasking on a UNIX operating system is reliable and well documented. This application's main program was designed to run run and manage its subprocesses on a UNIX operating system.

To simplify the operation of the program, a Graphical User Interface (GUI) was created. Buttons and sliders on this interface will allow the user to modify the `MarkerBroker` application for his/her requirements. This GUI was developed using the standard GUI for UNIX, which is the `Motif` style.

It is possible to obtain from the author a copy of the `MarkerBroker` program in a 'ready to run', 'executable' format for a very few operating systems. Most people will want to build the program themselves from the source code. They will need a C++

compiler and the standard UNIX development utilities, i.e. 'make'.

In summary, the requirements for a computer system to build the `MarkerBroker` application:

- A UNIX operating system such as SGI's IRIX or any standard Linux. All of these include support for the required TCP/IP networking.
- Support for Motif (OpenMotif) and the underlying X11 windowing system
- Unix development utilities, including the 'make' utility, a C++ compiler, and the Motif development header files and libraries.

4.2 Important features of MarkerBroker

`MarkerBroker` is created from software which defines three main features. All three of these features are shown in the figure below. The first is an sender interface where the user can select which combination of marker hardware he or she is trying to use, and where and how fast to send the data. The second is a set of very small 'sub' applications which are started as per the hardware combination. And finally, a block of memory which is shared between all components so that the sub applications can record their part of the data at one rate while the main application sends the whole data set at another rate.

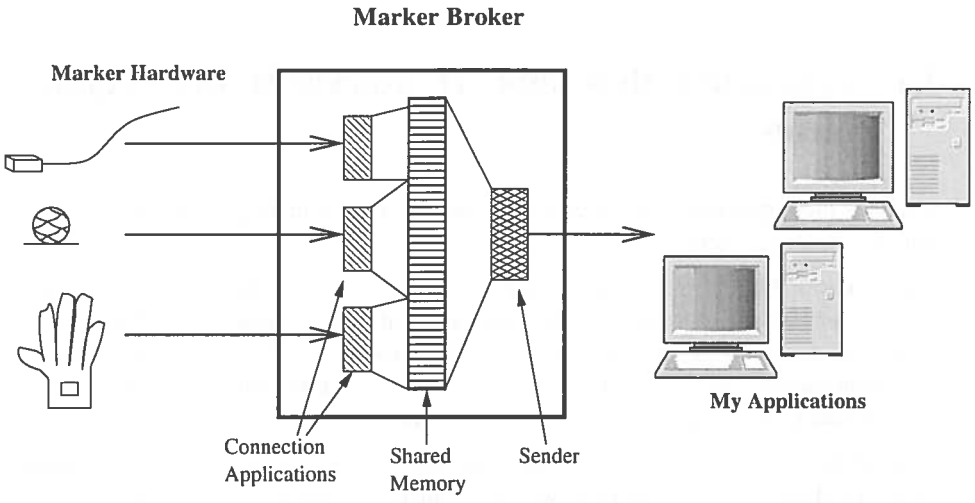

Figure 4.1: Main components of MarkerBroker

Small 'sub' applications (called 'connection' applications) are created for each type

of position tracking hardware. Each connection application is kept small and simple to reduce time for development, but existing code and vendor example code can be used if desired. Each application will be run as a separate process on the computer so that it can communicate with the tracking hardware at whatever rate is defined and/or required by that hardware. Every connection application just has to save the latest copy of its position data into the designated sub-section of the shared memory.

The main application is controlled by the user with a Graphical User Interface (GUI). The buttons and menus of this interface allow the user to define one (or more) destinations for the whole marker data set, and set the rate at which this data is retransmitted to the destination(s). Data is sent in a single, simplified format that should be easy for applications to read and use.

Since many different hardware combination could be supported, and since it could be time consuming to set all the destination data via the interface, a file is read by the `MarkerBroker` application to define all the initial parameters. This 'configuration' file is in a readable, standard, XML format. This file can be checked by other tools for completeness and consistency. Once the file has been read by the main application, all the required connection applications will be started, all the destinations will be set, and the send rate will be defined. The user can make changes via the interface if required, or just press the "Send" button to transmit all the position data.

4.3 How to use the MarkerBroker application

You start the `MarkerBroker` application by typing the name of the application in any 'shell' or 'terminal' window, or by double clicking on the application's icon when using your operating system's file browser. `MarkerBroker` will start, create its basic set of buttons, slider and menus. The result will be a window that looks like that shown in the figure below. There may be slight differences due to the text fonts and color schemes defined for your operating system.

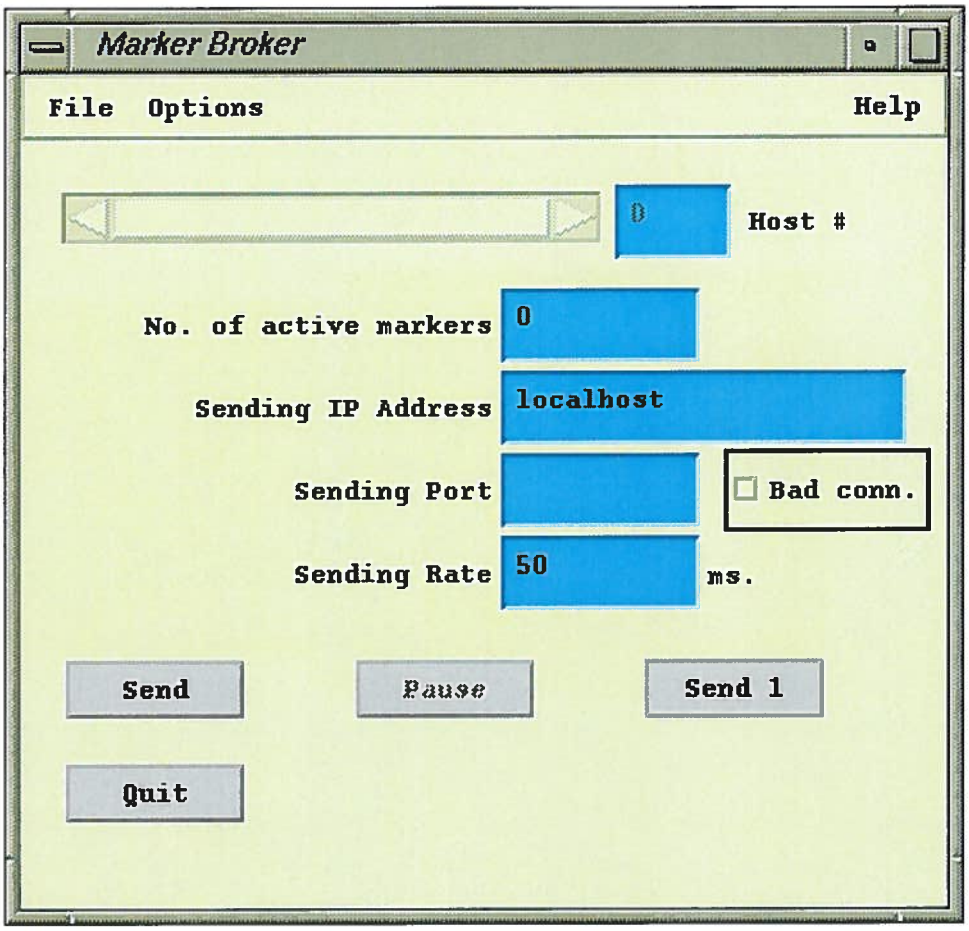

Figure 4.2: Main window of MarkerBroker

It is also possible, when starting the program from a terminal window, to start the program as described above, and then add the location and name of a configuration file on the same command line. `MarkerBroker` will automatically try to read the specified configuration file, and then set the application's appearance and behavior as defined by this configuration. Use this method to bypass the file selection sequence described below.

4.3.1 Main window controls

Once the `MarkerBroker` application has been started, you normally interact with the application by making selections from the application's main menu items. There are three main menus. Starting from right to left these are:

1. Help. This menu provides two choices. 'About' will display a small window to show the name of the application, its creation date, and the author's name and organization. The 'Instructions' option will use a web browser tool to display instructions on how to use the application, i.e. how to select configuration files and how to send data to remote applications.

2. Options. This menu allows you to turn on the display of two optional windows.

   - If you select the 'User Defined Marker' option it will cause a large window to be shown that contains sliders, labels and limits. Set the values of the labels and limits with the configuration file. Move the sliders to select six values for the user defined marker that is marker '0' in every data message sent. Click on the 'Dismiss' button in this window to remove this default marker display. You can also use the main window options menu to toggle off this default marker display.
   - If you select the 'Current Configuration' option it will show another window and display a condensed version of the active configuration file. Use the sliders, if necessary, to scroll around and view the configuration. Use the window manager controls to remove this window when required, or use the main window options menu to toggle off this extra display.

   Also provided on this options menu is another toggle option that will change how many messages will be printed on the terminal window where the application was started. If the 'Debug Mode' is "ON", then most application functions will print descriptive messages about their operation and current values. This option is used to test changes to the program source, and during the development of new connection applications.

3. File. This menu has several options. The 'Open' option will display a file selection window that will allow the user to navigate the file system and specify a configuration. The remaining options duplicate the function of main window buttons and are described below.

Normal operation would be to use the 'Open' option in the 'File' menu to select the desired configuration file. The appearance of the main window will then change.

When there is more than one destination defined in the configuration then the main window slider will be active. Use this slider to select the destination host whose address and port will be shown on the main display window. Each destination will be assigned a number, starting at zero. This number is shown in the field next to the slider.

Every configuration will define a number of position and orientation markers, even if it is only the '0' marker defined by the sliders in the optional window. The total number of markers will be shown in the field next to the label.

A destination is specified by a TCP/IP network address and a port number. Note: Think of the address as a city name and the port as a street number. Both values must be valid before you can send data to this destination. As you select destinations using the slider, the associated address and port will be displayed on the main window. You can make changes to these values if desired. When a destination address/port combination is not valid, or if the system cannot connect to this destination, then the 'bad connection' indicator will be "ON" whenever this destination's data is displayed. Users can modify the displayed destination address and/or port at any time that the `MarkerBroker` application is not sending. Changing a destination will cause the old connection (if one existed) to be broken, and a new connection attempted.

The 'Sending Rate' will show the time, in milliseconds, between messages when the `MarkerBroker` application is in automatic sending mode. This rate can be defined in the configuration file, but it can also be modified by the user.

Three buttons control the sending of marker data to the destination(s). When 'Send 1' is selected by the user, it will cause the current contents of the shared memory block to be read, a message to be formatted, and this message sent to all the valid destinations in the set of hosts. Select the button again to send another message.

You can also select the 'Send' button to automatically send data messages to all destinations at the rate specified on the main window. While in automatic mode, the 'Send 1' button is disabled and the 'Pause' button is enabled.

Stop the automatic sending by selecting the 'Pause' button. The flow of data messages will stop, and the 'Send' and 'Send 1' buttons will be enabled again.

If you select the 'Quit' button, all network connections will be closed, the connection applications will be stopped, the shared memory will be released, and the application will exit.

4.3.2 User defined marker

Every configuration defined for the `MarkerBroker` application will include a default position marker. Every data message transmitted by the application will contain this default marker as the first set of six values. The default marker data can be used to test or debug the remote application because its values can be exactly set by the user.

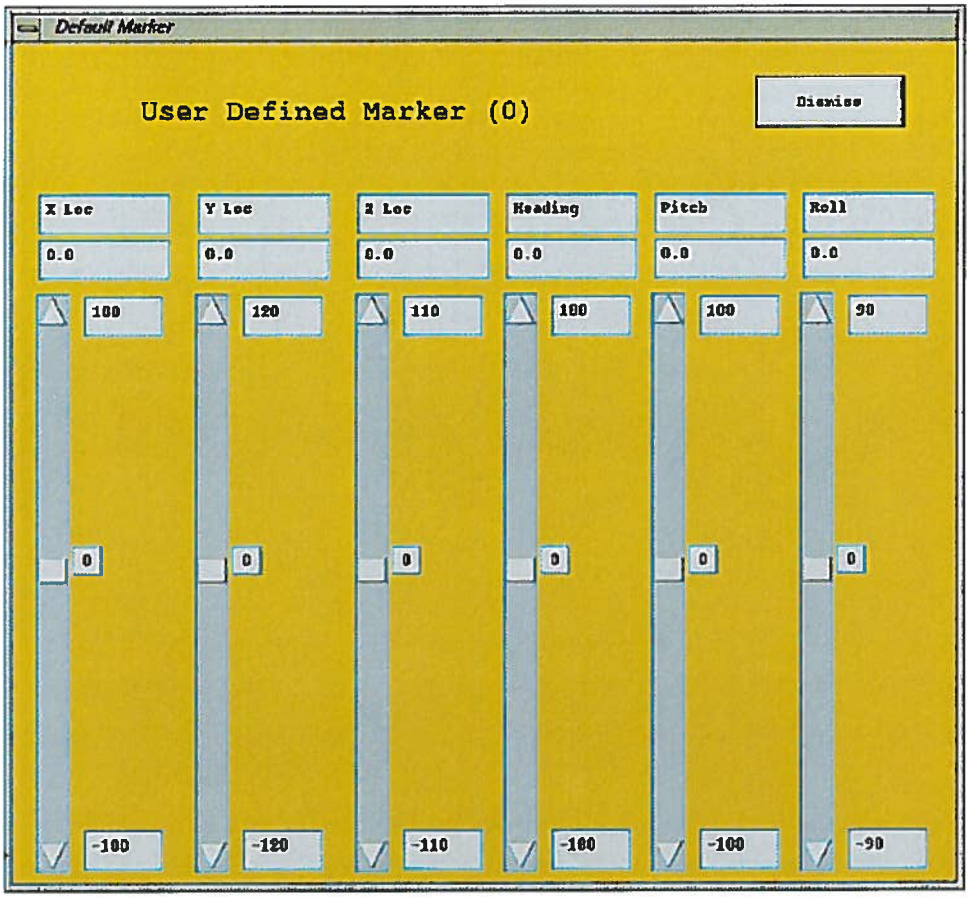

Figure 4.3: Default marker window

A default marker is defined by the position of six sliders in this window. Together these six values define one position and orientation, i.e. one six Degree Of Freedom (DOF) marker. Each slider has a label which can describe which DOF the slider has been assigned to. The current value for this DOF is shown below the label, and can be set by changing the position of the associated slider.

Each slider can move between a minimum value, which should be negative, and a maximum limit, which should be positive. The current values of both limits are shown beside the slider. A small '0' button is provided to reposition the slider back to a value of '0.0'.

All six sliders will have their labels, minimum value, and maximum value set by the required section in the configuration file. These values cannot be changed after the configuration file has been read.

A 'Dismiss' button on this window can be used to remove the window from the display. Every data message sent by the `MarkerBroker` application will contain a default marker whose values match the current settings of the six sliders even when the default marker window is not visible.

4.3.3 Configuration window

A configuration file provides important, required information to the `MarkerBroker` application. It is necessary to select a configuration file before the application can connect to real marker hardware. Without a configuration file the application will not be able to send any data. How do you verify that the user has specified the file required by his/her hardware and remote applications? Users can toggle on the display of the configuration file display window.

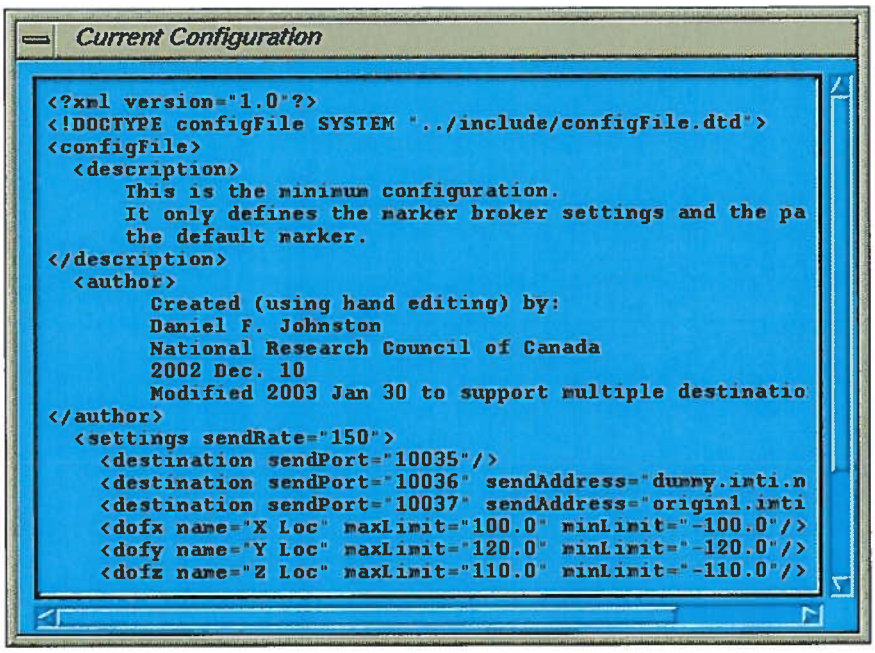

Figure 4.4: Current configuration window

This window will simply allow the user to scroll and read the current configuration. The appearance of the file will probably be different than the one supplied by the user, but the contents will be the same. It is not possible to change the configuration using this display window.

Remove the configuration display window by changing the 'options' menu selection in the main window.

4.4 Format of the sent marker data message

Position marker data is sent by the `MarkerBroker` application using a very simple format. Each message is sent as a TCP/IP Datagram socket to every valid destination address and port on the network. Datagram network messages are fast and easy to read by applications, but their delivery is not guaranteed. Very poor networks may drop some data messages, but this is very rare on modern equipment.

The contents of the message is all printable characters so that there will never be a data format compatibility problem. Every integer and floating point value is represented as a set of characters (sign, numbers and decimal point) separated by a space from the next value. There are no other characters or codes in the message.

The very first value is one or more characters which define an integer. This integer value will equal the total number of six DOF markers are provided in this message.

Each marker is defined by six floating point values. The six values are separated from each other, and from the next value, by a space. After the end of the last value in the last marker, the message will end.

An example message would look like;

"5 0.0 0.0 20.5 -3.56 0.0 -45.0 10.0 10.001 -10.001 0.0 90.0 23.55001 18.2345 -30.65 5.246 -90.0 0.0 10.005 254.006 -27.995 0.0034 0.0 -180.0 0.0 -8.554 -175.92 90.225 -0.5528 1.002 0.0007".

Note: The double quotes show in this sample are not part of a real message.

4.5 What might go wrong and why

- Do not try to send marker data when none of the destinations appear to be valid. Correct the configuration file and restart the application, or edit the destination address information on the application's window, before trying to send data.

- Excessive address and port destination errors can cause problems for the `MarkerBroker` application. Try to specify only valid connection data. Try not to change the configuration file several times in one use of the application.

- A destination connection can be invalid if the address is not valid (typing error in

the name or number, or non-existing address), or the address cannot be resolved by your network management tools (DNS).

- A destination connection can be invalid if the port number is out of range for that remote computer (range of valid TCP/IP ports is computer system specific), or the port number is already used by other applications.

- A destination connection can be invalid if there is no network connection between the computer which is running the `MarkerBroker` application and the remote destination. Either system could be unplugged from the network, or some network hardware between them may be blocking the flow of data.

- It is possible for the application to fail to start if the file used for a shared memory block already exists. Check for a file called "vectorMemory.shm" in the system "/tmp" (or "/var/tmp") directory and remove it before trying to start the application again.

- If a connection application fails to start (bad network connection, serial line parameters wrong, non-functioning position marker hardware) then the marker values assigned to that device will be at their default zero value.

- If a connection application stops reading data from its sensors (hardware or software error) then the data in shared memory and the data transmitted for that block of markers will remain at their last data values.

Chapter 5

Configuration File

This chapter describes the configuration files, their components, and their options.

It is the configuration file, once read and processed by the main `MarkerBroker` application, that creates the required number of connection applications, and that creates a network connection to each destination to receive data. The type of position tracking hardware that is used and the number of markers in each hardware connection is defined by the author of the configuration file using the options listed below.

Several complete configuration files are listed in an appendix of this document. These can be found in the 'configs' directory when user has a copy of the `MarkerBroker` application.

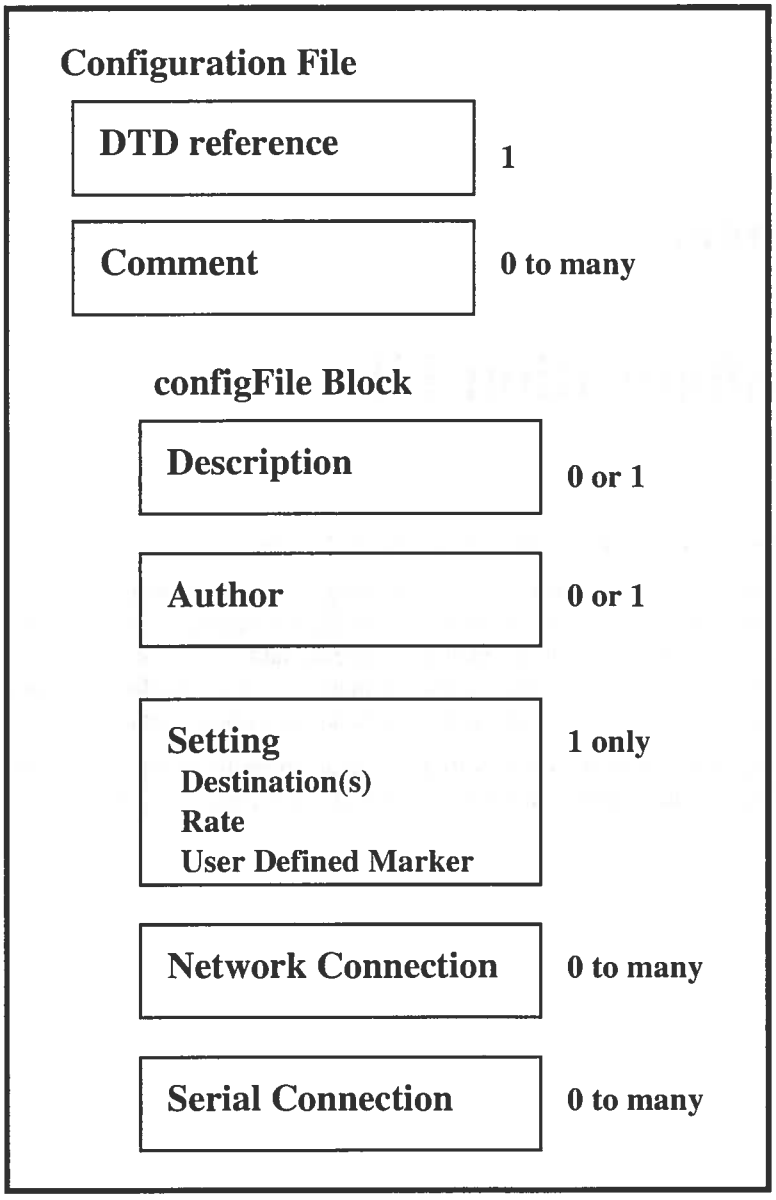

Figure 5.1: Configuration file structure

5.1 General features of configuration files

Every `MarkerBroker` configuration file will *probably* include one of each of the elements shown in the figure above. Some options, as described below, can only exist once or not at all, some must always exist, and some can exist zero to many times. This is specified exactly in the DTD description included by every configuration file. This DTD can be used by XML parsing tools (not described in this document) to verify new configuration files. This DTD file should only be modified by someone who is very knowledgable about XML and DTD.

It is possible to add comments to any configuration file. Comments are (usually) descriptive text that is ignored by the computer applications that reads the files. They are added only for human readers. Comments can be positioned anywhere outside the 'configFile' block using the standard XML comment format, i.e. you start a comment with the characters

```
<!--
```

and end a comment with the characters

```
-->
```

The start and end codes, and everything in between, will be ignored by any XML reading program. See the configuration file described below for an example.

5.2 Header section

Every configuration file must start with the same few lines at the top.

```
<?xml version="1.0"?>
<!DOCTYPE configFile SYSTEM "../include/configFile.dtd">
<!--
  Please do not delete or change this file. It is used by
  the documentation for the 'marker broker' software.
-->
```

The first line defines the minimum version of XML which is needed to read this configuration file. We only specify the minimum, version 1.0, because there is nothing very complex about this document. This line is required.

The second line includes the Data Type Description (DTD) file. Every configuration file must include this. It defines the document structure and provides the default values for some parameters. This line is required.

Next comes a comment, as described above, which is ignored by the configuration file reading code. Comments are optional, and can be placed anywhere outside the 'configFile' definition.

Finally we start the 'configFile' block which is the main feature of our configuration file.

```
<configFile>
```

5.3 Description and Author sections

Every `MarkerBroker` 'configFile' block will *probably* include a description of what position tracking hardware and destinations are supported by the file. Since this information might be of some use by an application which reads this file, the information is defined within a 'description' section. It is not required that a configuration file have a description, but if one is supplied it can be located anywhere in the block. There can be at most one (1) description section per block. This information is not used by the `MarkerBroker` application.

```
<description>
      This is a very basic configuration for testing.
      It defines the full set of broker settings, and then adds one set of
      network markers (Motion Analysis HTR2) and one set of serial line
      markers (Ascension FOB).
</description>
```

Another section has been defined to hold information in an 'author' section of the 'configFile' definition. You should place here the name of the person who created this file, his/her organization, and any contact information. Again, it is not required that a configuration file have an author section, but if one is supplied it can be located anywhere in the block. There can be at most one (1) author section per block. This information is not used by the `MarkerBroker` application.

```
<author>
        Created (using hand editing) by:
    Daniel F. Johnston
        National Research Council of Canada
        2002 Nov. 20
    Modified 2003 Jan 30 to support multiple destinations
</author>
```

5.4 Setting section

This subsection of the 'configFile' will allow the end user to set a default rate for automatically sending the data to destinations. This rate would be defined as a whole number of milliseconds between data sets.

```
<settings
   sendRate="80.0"
```

If a rate value is not supplied, a default value will be set by the DTD file.

Also part of the 'setting' section is the subsections for destinations and user defined marker.

5.4.1 Data destinations

A 'destination' is made up of the required information to create a TCP/IP network connection to that computer. You need an address and a port number. If you do not supply an address, the DTD file will supply a default address which points to the computer you are running the `MarkerBroker` application on. The port number is always required.

```
>
<destination
   sendPort="10035"
   />
```

For most destinations you need to supply both the address and the port.

```
<destination
   sendPort="10036"
   sendAddress="dummy.imti.nrc.ca"
   />
<destination
   sendPort="10037"
   sendAddress="origin1.imti.nrc.ca"
   />
```

Notice that the 'dummy' address in this example will probably fail, i.e. it will not be possible to establish a network connection because the address is not valid. The user could correct the address and establish a connection by fixing the address using the application interface after the `MarkerBroker` program has read this configuration file. `MarkerBroker` will not send data to an invalid destination.

5.4.2 User Defined Marker

There is always at least one six value marker provided by every block of data sent by the `MarkerBroker` application. This marker's values are set by the position of sliders in the application's optional 'user defined marker' window. It is required that range values be specified for each of these six sliders. Their labels can also be set, but a default label will be supplied if they are not.

```
<dofx
   name="Loc X"
   maxLimit="102.5"
   minLimit="-86.45"
   />
<dofy
   name="Loc Y"
   maxLimit="120.0"
   minLimit="-97.3"
   />
<dofz
   name="Loc Z"
   maxLimit="200.0"
   minLimit="-150.0"
   />
<dofh
   name="Orient H"
   maxLimit="110.0"
   minLimit="-110.0"
   />
<dofp
   name="Orient P"
   maxLimit="45.0"
   minLimit="-45.0"
   />
<dofr
   name="Orient R"
   maxLimit="90.0"
   minLimit="-90.0"
   />
```

Every one of the six sliders must have a minimum value and a maximum value specified. The minimum value should be a negative value. The maximum value should be a positive value. Users can define a short text label to be assigned to each slider, or a default label would be supplied by the DTD.

There are six sliders required, and it is an error not to supply all six in a configuration file. The six degree-of-freedom (dof) sliders (x, y, z, h, p, r) can appear in the 'setting' section of the configuration file in any order.

After the last of the destinations and/or the end of the user defined marker data, the 'setting' section is terminated.

```
</settings>
```

5.5 Network connections

There is a document section where the user can describe any network connections. This section is only needed if the user will be reading data from a position tracking device connected to the computer network. He/she would create a 'network' section for each device they will be reading from.

```
<network
   readPort="10032"
   numbMarkers="23"
   type="test_net"
   />
```

Each network section must contain the information required to establish a link to that hardware. Several parameters can be supplied in this section;

- readAddress - The TCP/IP network address for the supplier of position data. This parameter is required in general, but will default to the local computer if not supplied.
- readPort - The network port number used by the supplier of data. This parameter is required.
- numbMarkers - The total number of position markers that will be defined for this source is specified in this required parameter. If you specify more markers than the hardware can supply, the connection application should provide zero data for the rest. If you ask for fewer markers than there is position sensors in the hardware, the connection application should only copy the data from the designated number
- markersType - Position tracking data is usually supplied as three position values (x, y, z) or six values with position and orientation. If the connection application is set to '3dof' then the 3 orientation values will always be zero. This parameter can only be set to '6dof' or '3dof'. It will default to '6dof'.
- type - Each type of position tracking hardware must have a unique connection application written for it. The 'type' of the connection is used to identify which application the main program must start. The parameter must be set to a type value that is listed in the DTD document, and which matches the name of a supplied connection application. One 'type' parameter must be supplied.
- scale - A translation scale value can be supplied. This number, with a default value of 1.0, will be multiplied against the first three data values from each position marker. This value will scale the translation part of a marker, not the orientation.

5.6 Serial line connections

There is also a document section where the user can describe any serial line connections. This section is only needed if the user will be reading data from a position tracking device connected to the computer using a serial line or 'COM' connection. he/she would create a 'serial' section for each device they will be reading from.

```
<serial
   line="/dev/ttyd3"
   baud="9600"
   readRate="150.0"
   numbMarkers="3"
   type="test_serial"
   />
<serial
   line="/dev/ttyd2"
   baud="19200"
   readRate="150.0"
   numbMarkers="1"
   type="fob_ea"
   />
```

Each serial line section must contain the information required to establish a link to that hardware. Several parameters can be supplied in this section;

- line - Which serial line to use for communication? This parameter is required. Values such as "COM1" or "/dev/ttyd2" would be valid on some computer systems.
- baud - Communication over a serial line is specified using an older measurement of "bits per second" or "Baud rate". 9600 baud would be the same as 960 characters per second. The parameter defaults to a value of 19200 baud.
- readRate - The user can specify the time in milliseconds between reading data from the hardware. If the specific hardware defines its own data rate, then this parameter is ignored by the connection application.
- numbMarkers - This parameter has the same meaning as described in the network section.
- markersType - This parameter has the same meaning, and accepts the same two values, as described in the network section.
- type - This parameter has the same meaning as described in the network section. The type names in the DTD file and thus the names of the connection applications will be different for serial connections, so the set of valid parameter values will be different.
- scale - A translation scale value can be supplied. This number has the same meaning as for the network connections.

5.7 The tail of the configuration file

Every configuration file must contain a line to terminate the 'configFile' section, and thus finish the file.

```
</configFile>
```

Chapter 6

MarkerBroker install and build

This chapter describes where you can obtain a copy of the `MarkerBroker` application program software, and how to build the application once you obtain it. A few options in the build process can adapt the software to changes in the on-line help file viewer.

6.1 Computer systems which support MarkerBroker

All the source code for the `MarkerBroker` application, and all the supplied set of connection applications, was written using the C++ computer programming language. Any target computer system should have a set of development tools that include a C++ compiler.

The `MarkerBroker` application was designed as a set of multitasking programs running in a UNIX style of operating system. It relies on the UNIX 'fork' and 'exec' commands, as well as the UNIX style of 'signals' to control the connection processes. Therefore the application will only be easily supported on a computer which is running a UNIX or Linux operating system.

Interaction with the main application is done with a Graphical User Interface (GUI) written by the 'builder' tool in the `Motif` style. This style should be available on every UNIX-like computer system.

6.2 How to obtain the source for MarkerBroker

The source code for the `MarkerBroker` application is copyright by the National Research Council of Canada. All requests for a copy of this source code must be made via the "Director of Research" for the author's NRC Institute. File transfer via the computer networks or a copy of a CD can be arranged. Either method will provide source code files, a pre-compiled application specific to a particular computer system, or both.

The author used a limited number of operating systems to test the `MarkerBroker` application during development. These systems are retested after changes that add new features and correct problems. If one of these test environments match the user's requirements, then the pre-compiled application can be used directly.

The more general situation requires that the user recreate the application by compiling the source files to create 'object' files, and linking these object files to create the application program. There are many source files. Obtain a copy of all the files, arranged in the directory structure described below, and then rebuild the application using the suggested procedure.

Remarks:

All the source code produced by the author is contained in the directory structure shown below. The application must also be linked with an XML parsing library, which the author did not write. Build procedures supplied with the application will try to link with a common public library called `libXML2`, but any similar XML parsing library could be used. A change in this library will require a change in the build process and a number of changes to the source code for the classes which read portions of the configuration file.

6.3 Project directory structure

All the source files for the `MarkerBroker` application have been grouped and organized as shown in the figure.

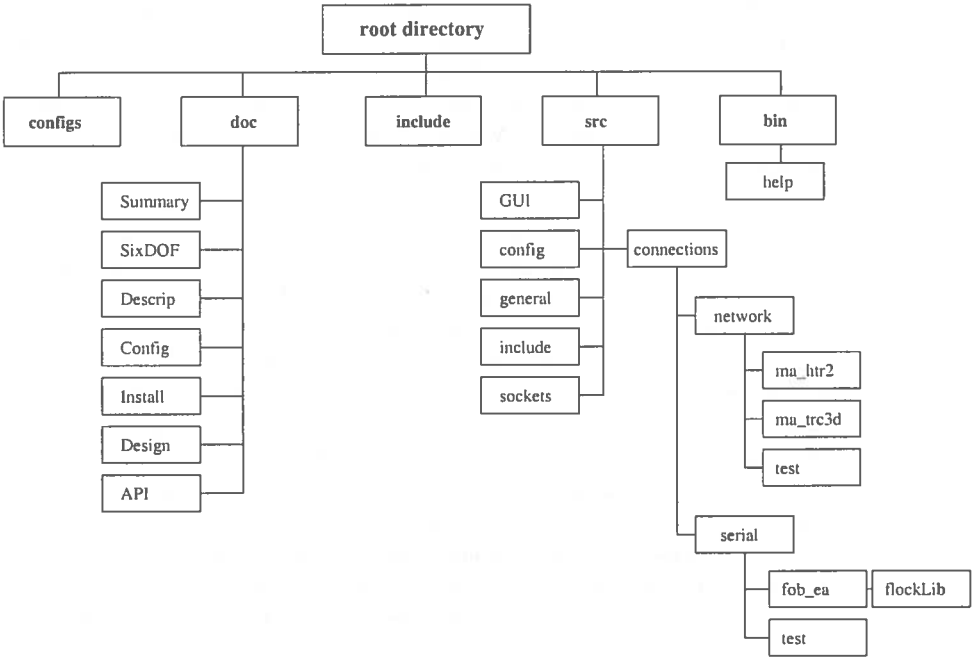

Figure 6.1: Directory structure of MarkerBroker

The following files are found in each directory;

- root - The top of the directory structure contains only the sub-directories, a 'Makefile' which will build everything (all the applications and documentation), and a file called "setup-env". This last file would contain a few definitions that the application needs to build and run - in this case there are only a couple of settings needed when trying to rebuild the documentation.

The first level of subdirectories:

- include - This subdirectory contains one file which is needed by all the XML configuration files. They all include a "Data Type Description" (DTD) file which defines the required configuration structure, all the required parameters, and all default values. Do not change this file unless you are familiar with XML and are prepared to change all the existing configuration files and all the source that parse them!
- configs - This directory contains a number of XML configuration file documents. Each one will specify a different combination of marker hardware and destination settings. Some were created for testing during the development of the `MarkerBroker` application. Some are for configurations that were used for demonstrations of the application with real tracking hardware.

- bin - All the applications built for the `MarkerBroker` project are stored in this area. More correctly, a copy of each executable program that is built by the linker is placed in this subdirectory. This location is important because the main application will look for the connection applications in the same directory where the main application was started. A 'help' subdirectory contains the on-line help instructions and the images needed for this help file.
- doc - All the files needed to build the documentation for this report are found in this directory and its subdirectories. Inside the 'doc' directory is the text file used to create the introduction chapter. Each sub-directory contains the text and images needed to create the remaining chapters, e.g. the subdirectory 'Descrip' will contain the `Descrip.doc` text file and images used to create the chapter which describes how to use the `MarkerBroker` application. There are also a number of files which are used by the 'doxygen' documentation generator tool, and which are not described here. One special subdirectory of 'doc' is API. It is here where the documentation is created from all the source code. It is here where the group names are defined that determine which classes and methods are described together as 'modules' in the source code documentation. One 'API' subdirectory will therefore create many chapters of the documentation.
- src - All the source files needed to create the main application and the connection applications is contained here. These will be described more completely below.

The main application's source files;

- GUI - A Motif user interface builder tool, Builder Xcessory from ICS (`bx50`), was used to build the GUI source code. The file `main.uil` contains the layout, color, and interaction descriptions as defined by the author using this tool. Use `bx50` and this file to reload the building environment and thus make any corrections or changes. Then you select the 'generate C++ code' option on the builder tool. It will create new versions of most of the files in this directory. New files will merge with existing files to preserve user changes. Not all the files produced during the code generation are used by the `MarkerBroker` application. The computer generated 'makefile-C', for example, is replaced by the author created 'Makefile' to be consistent with the build options in all the other 'Makefiles'.
- include - Some of the computer generated files in 'GUI' will contain 'include' statements. These lines, at compile time, will bring in sections of code from other files to extend the data definitions section of the main window class, for example. The files which are included by the GUI source files are all found in this subdirectory. These files are fully documented. The files which 'include' them (found in the 'GUI' directory) are not.
- general - Almost all the class methods (i.e. internal class functions) which are mentioned in the expanded GUI class files are defined in the source files in this directory. There are methods defined for the main window class, and a few for the value slider class.
- config - One important method for the main application is to start reading the configuration file and setting the application based on its contents. This directory contains the source code to do this reading. Since the configuration breaks

down into sections such as destination, network connections, and serial connections, the parsing of configuration is broken into different classes. Each class file is responsible for reading one section or subsection of the configuration. All the source files and classes for reading the configuration file are found in this directory.

- sockets - This application needs to create/delete/configure TCP/IP UDP sockets to communicate the shared data to all destinations. Many connection applications also need this type of network connection. The source files for a 'socket' class were borrowed from a previous project, but these had to modified to fit the `MarkerBroker` application. All the source files for the socket class are found in this directory.
- connections - All the connection applications that are provided with the applications are found in subdirectories of this one.

Connection application subdirectories;

- network - Position tracking hardware which communicate using TCP/IP sockets are monitored and read by connection applications defined in subdirectories of this directory.
  - test - The simplest application that just connects to the block of shared memory and then writes data into it. Change this source file to change which data is written. No network connection is actually make by this code. It is used as a dummy source of data for testing.
  - ma_htr2 - This application will connect to a passive, optical tracking system produced by Motion Analysis Corp. ('ma'). This system can track complex machines and people movements, and provide the data on joint positions and angles. Such data is provided from this system as socket messages in a format the company calls 'htr2'.
  - ma_trc3d - This application will connect to a passive, optical tracking system produced by Motion Analysis Corp. ('ma'). The fastest and simplest data that can be transmitted over sockets by this system is the 3D position data for every marker that can be seen. The company calls this data format 'trc'. Only the 3D position information is provided, so the application records zeros for the three orientation values of each marker.

- serial - Position tracking hardware which communicate using serial line connections are monitored and read by connection applications defined in subdirectories of this directory.
  - test - The simplest application that just connects to the block of shared memory and then writes data into it. Change this source file to change which data is written. No serial line connection is actually make by this code. It is used as a dummy source of data for testing.
  - fobea - A very common type of magnetic tracking is the "Flock of Birds" (fob) system from Ascension Technologies. It is available in the older

'flock' hardware and the new 'motionstar' system which can support many more sensors. Both systems can operate in 'extended addressing' ('ea') mode to select sensor channels. An NRC modified library is used to communicate with any version of this hardware. The source code for this library is provided in a subdirectory called 'flockLib'.

6.4 General comments on the building process

Almost every directory and subdirectory in the `MarkerBroker` application will also contain a file called '`Makefile`'. A standard UNIX software development utility, called 'make', will use this file to automate the process of compiling source code into object files,and linking object files and libraries into executable programs. Directories which do not contain source files can still contain a `Makefile`, which will be responsible for building everything in the subdirectories below it. Thus, the `Makefile` in the top directory would rebuild the main application, all connection applications, and all the documentation. This `Makefile` is listed below.

```
# The source directory stuff MUST be built first
# The remaining directories can be built in any order.
DIRS = \
    src \
    doc \
    $(NULL)

default:
    @for i in $(DIRS); \
    do \
        echo "\nmaking $$i ($@) in $(PWD)"; \
        cd $$i; $(MAKE) -f Makefile $@; cd ..; \
    done

debug:
    @for i in $(DIRS); \
    do \
        echo "\nmaking $$i ($@) in $(PWD)"; \
        cd $$i; $(MAKE) -f Makefile $@; cd ..; \
    done

clean:
    @for i in $(DIRS); \
    do \
        echo "\nmaking $$i ($@) in $(PWD)"; \
        cd $$i; $(MAKE) -f Makefile $@; cd ..; \
    done

clobber:
    @for i in $(DIRS); \
```

```
    do \
        echo "\nmaking $$i ($@) in $(PWD)"; \
        cd $$i; $(MAKE) -f Makefile $@; cd ..; \
    done

depend:
    @cd src; $(MAKE) -f Makefile $@; cd ..
```

Several main build options, called 'targets', are defined in this and every other Makefile in the entire application.

- default - If no other make option is specified, i.e. just the command "make", this default target will build an optimized version of its code. This would be the smallest version and/or the one that would execute the fastest. As there is no source files in this directory, this option will just cause an optimized 'make' to run in the 'src' and 'doc' subdirectories.
- debug - This option, selected with the command "make debug", will also build all the applications and documentation, but now the applications will include extra information which can be used when searching for errors and verifying operation during development. Note: There is no difference between 'default' and 'debug' when building documentation.
- clean - Many of the temporary files created during the build process can be removed with a "make clean" command. This will also remove all the object files in source directories.
- clobber - When the command "make clobber" is given, not only will all source directories be 'cleaned', but all applications produced by the build process will be deleted. All temporary files and computer created subdirectories will also be emptied and deleted.
- depend - This target is not normally requested by developers. It is selected automatically if needed during the "make" or "make debug" build process.

This top level `Makefile` was used frequently by the author to recreate all the application and documentation for this project. He only had to issue two commands;

1. "source setup_env" - This command would include the definitions in the 'setup_env' file for use by all subsequent commands. This command only has to be used once, and only if you intend to rebuild the documentation. Note: This command will only work with the supplied file if you are using a UNIX 'tcsh' environment.
2. "make" - Build all the source code into applications, and then rebuild all the documentation from the text and the source files.

As it is time consuming to rebuild the documentation, it is faster to issue the "make" command in the 'src' directory after any corrections to the source files. Once the source and the applications are fully tested you can rebuild the documentation by issuing the "make" command in the 'doc' directory.

6.5 Building the MarkerBroker applications

Most of the source files provided with the `MarkerBroker` project are required to build the main application. Since this project is written using the C++ computer language, the code will define the object and classes (and data and methods) which contain the logic and functionality of the application.

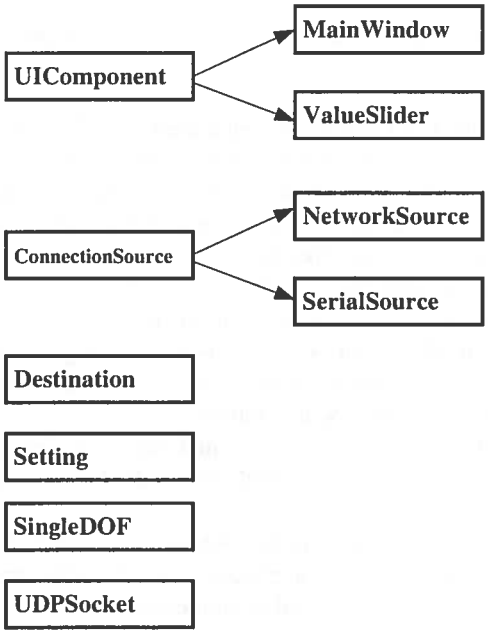

Figure 6.2: Main classes defined for MarkerBroker

Set your default location to be the 'src' directory. Look at the `Makefile` in this directory to note the sequence defined for which subdirectories are built before the main application. Any order is valid, as long as the source code in the subdirectories listed before 'GUI' is compiled before the code in this directory.

Once the 'GUI' directory's code is compiled, the `Makefile` in that directory will try to build the main `MarkerBroker` application. It will need to link to the object files in other subdirectories to create this application, so these object files must be created first. Build procedures supplied with the code (i.e. in the 'src' directory) will make sure a valid procedure is followed, but you can 'make' in the individual subdirectories as needed and then 'make' in the GUI directory to update the application.

Connection applications can be built in any order because they are designed to be small and independent. Supplied `Makefiles` will build them after the main process.

After any main or connection application is (re)built it will be copied to the 'bin' di-

rectory so users can run the program.

6.6 Building the project documentation

All documentation for the project, including this report, is built from simple text files and from special comments added to the source code. 'Doxygen' is a software documentation tool that is used to to convert these text files, comments, and source code into the desired documentation formats. This software tool is freely available, can run on all UNIX and PC platforms, and is often included with many Linux distributions.

Inside the 'doc' directory are many files that are required for creating the documentation. There is also a `Makefile` which will recreate the documentation for you once you have the `doxygen` software installed on your computer.

It can take several minutes to recreate the documentation for all the computer source code for this project. Examine the `Makefile` in the 'doc' directory to see how to rebuild only those parts of the project documentation which have changed.

Two of the most frequently used versions of the documentation produced by the `doxygen` tool are 'html' and 'latex'. The first is documentation in a format that can read by anyone using a standard WWW browser, such as `netscape` or `explorer`. You can read text, view images and figures, and used 'links' to jump to other chapters and source code. Just point a browser at the 'index.html' file in the 'html' subdirectory that is created by `doxygen`.

A second important format for the documentation is found in the 'latex' subdirectory of 'doc'. A common UNIX software tool can convert the files found in this directory, and its subdirectories, into documentation suitable for high quality printing.

The documentation is also produced in a 'Rich Text Format' (rtf) version which can be read into most PC word processing systems.

When the source files for the `MarkerBroker` software is obtained as described above, the most recent documentation will also be provided in all formats. These prebuilt documentation files can be used even if the user does not have access to a copy of the `doxygen` tool.

6.7 Additional build options

Inside the `Makefile` in the 'GUI' directory there is a line which tells the build system which file is used to provide on-line help for the `MarkerBroker` application. This file is provided in 'html' format. Another line will define the local program to display this help file. As supplied by the author this is 'netscape', but many Linux systems

would need something like "konquerer" or "/usr/bin/mozilla". Edit these lines to build the application with the help command and file required for the target system.

Chapter 7

MarkerBroker design considerations

This chapter describes very briefly the key design concepts for the `MarkerBroker` application. These are shown as a list of 'ideas' in point form, and are not listed in any order of preference or importance.

This list of design ideas is intended as a section for those who need to maintain or extend the application. It will also serve to remind the application's authors about the reasons some features were put into the program, and how the programs components are supposed to fit together. For these reasons this chapter is placed close to the chapters which describe the source code of the software in detail.

A couple of ideas about future directions for this software are included at the end of this chapter.

7.1 Requirements for a Marker Broker

- Each type of hardware will have unique communication requirements
  - There could be a serial or network (TCP/IP) connection.
  - Hardware control and communication is vendor specific.
  - Data transfer rates are hardware specific.

- We need to create and destroy sub-processes (connection applications) as required by the user-selected configuration file. We will use a UNIX operation system for the application because the multitasking is well understood and reliable.

- The application was written in the C++ computer language. It is intended that this computer language choice will make the source code easier to understand and to modify.

- The utility program 'Doxygen' was used to create the documentation for the source code. It will create class and method descriptions from special comments imbedded in the source code. It will also produce documentation for all structures and global variables, it can sort methods and functions into author defined categories, it can produce an index of all function and variables, and will produce documentation in several common formats. It is not necessary to have a copy of `Doxygen` to create the application program from the source code, but you do need it to recreate the documentation.

7.2 General operation of MarkerBroker

- Encapsulate each hardware connection with a specific 'connection application' program. These are designed to be as small and simple as possible to minimize development complexity and time.

- Create a control program to start the necessary connection application. Provide a block of shared memory to all parts, i.e. the main program and the connection application. The connection applications store their data in a designated part of the shared memory block. The main application will send the total data set out in a very simple format. Data is sent via a network connection, one set at a time or at a defined rate.

- Use a configuration file to adapt the operation of the control program to the specific hardware and application requirements. The configuration file can be easily edited by the end user. A standard XML document format is used for this file. XML document tools can verify the contents and structure of these configuration files.

- The main application will start the connection applications, pass data to them, and stop them when the connection requirements change. This requires reliable support for multitasking, and a form of multitasking known to the author. The UNIX system of multitasking was selected.

- It is possible to use common software tools to automatically 'configure' a software project and its build sequence to match a variety of computer hardware and operating systems. This does add complexity to the software development, the build process, and the documentation. It was decided **not** to use these tools in

this project. Instead, a set of 'Makefiles' was supplied that should be easy to change for any different UNIX environment.

7.3 Specific to the main application

- All the source code is fully documented using the tool 'doxygen', except the code that creates and operates the Graphical User Interface (GUI). The GUI was written using a commercial 'GUI Builder' tool called "Builder Xcessory Version 5.0" from ICS. Source code is created from the GUI description by this tool. There are places in the source code where you can add comments and your application specific code. These locations do no support the comments required by the 'doxygen' documentation tool. So, the GUI source code is not very well documented. In several places the user created source code is placed into separate files and 'included' into the GUI source. These separate files **are** documented with 'doxygen'.

- The shared memory could be created as a "memory mapped file" or as a block of physical ('core') memory. The latter is faster. We chose the former, because it is fast enough for the application (disk caching helps) and it is much easier to clean up the shared memory if the application fails for some unexpected reason. This was especially important during the development of the application.

- The source for the shared memory vector class is all contained within the 'header' file. There is no other file, i.e. no '.C' or '.cpp' file. This was deliberate. It means that any application or library that wants to use this class will just have to include the header and will have all the information for the compiler to instantiate any form of the template.

7.4 Specific to connection applications

- Every connection application must connect to the shared memory for writting data.

- Every connection application must respond to the standard set of command line arguments that are passed on from the main application.

- Every connection application must connect to the network or serial line hardware and get data from the device at a rate that is hardware specific.

7.5 Advantages of using MarkerBroker

- It is easier to write applications that use `MarkerBroker` because it is easy to read and parse the data values.

- Data from several position tracking systems can be read in a single `MarkerBroker` message.

7.6 Disadvantages to using MarkerBroker

- For some sources of position data, reading the data from `MarkerBroker` will be slower than the best possible data transfer from the system alone. This will mean slower application responce to changes in object position, i.e. increased "latency", than would be possible if the application connected to the system directly.

7.7 Future directions for MarkerBroker

- Many vendors of position tracking hardware now supply example code and applications for the MS-Windows operating system. It would be nice to have a version of `MarkerBroker` for that class of computers. This would mean a new GUI, a new way of multitasking, and new software for socket connections.

Appendix A

DTD and Example Configuration Files
(Required DTD file and selected examples)

Source files included in this appendix:

1. configFile.dtd: The required "Data Type Description" file that must be included with every XML configuration so that the application can verity the structure and contents of the configuration.

2. minimumConfig.xml: The simplest example of an XML configuration file for MarkerBroker. It provides only the required information and thus the application will send only the default marker data to the local computer.

3. testConfig.xml: A configuration files which contains all the required and optional features of a MarkerBroker configuration file. This file is described in detail in the documentation.

4. demoTRCConfig.xml: This is a configuration file used in a working demonstration. It will read network data to obtain 3D position data (11 markers) from a Motion Analysis tracking system.

```
<!--

  This is the Data Type Description file for the configuration
  files used by the 'marker broker' software.
-->

<!ELEMENT configFile ( description?, author?, settings, network*, serial* )>

   <!ELEMENT description  (#PCDATA) >

   <!ELEMENT author       (#PCDATA) >

<!ELEMENT settings ( destination+, (dofx, dofy, dofz, dofh, dofp, dofr)? )>

   <!ATTLIST settings
      sendRate    CDATA "50.0"
      >

<!ELEMENT destination EMPTY >
   <!ATTLIST destination
      sendPort    CDATA #REQUIRED
      sendAddress CDATA "localhost"
      >

<!ENTITY % defPosLimit "maxLimit  CDATA '100.0'
                        minLimit  CDATA '-100.0'">
<!ENTITY % defAngLimit "maxLimit  CDATA '180.0'
                        minLimit  CDATA '-180.0'">

<!ELEMENT dofx EMPTY>
   <!ATTLIST dofx
      name      CDATA "Pos X"
      %defPosLimit;
      >
<!ELEMENT dofy EMPTY>
   <!ATTLIST dofy
      name      CDATA "Pos Y"
      %defPosLimit;
      >
<!ELEMENT dofz EMPTY>
   <!ATTLIST dofz
      name      CDATA "Pos Z"
      %defPosLimit;
      >
<!ELEMENT dofh EMPTY>
   <!ATTLIST dofh
      name      CDATA "Ang H"
      %defAngLimit;
      >
<!ELEMENT dofp EMPTY>
   <!ATTLIST dofp
      name      CDATA "Ang P"
      %defAngLimit;
      >
<!ELEMENT dofr EMPTY>
   <!ATTLIST dofr
      name      CDATA "Ang R"
      %defAngLimit;
      >

<!ELEMENT network EMPTY >
   <!ATTLIST network
      readPort    CDATA #REQUIRED
      readAddress CDATA "localhost"
      scale       CDATA "1.0"
      numbMarkers CDATA #REQUIRED
```

```
67          type          (ma_htr2 | ma_trc3d | fob_net | test_net) #REQUIRED
68          markersType (6dof|3dof) "6dof"
69          >
70
71    <!ELEMENT serial EMPTY >
72       <!ATTLIST serial
73          line          CDATA #REQUIRED
74          baud          CDATA "19200"
75          scale         CDATA "1.0"
76          numbMarkers CDATA #REQUIRED
77          readRate      CDATA "120.0"
78          type   (polaris | fob_ea | test_serial) #REQUIRED
79          markersType (6dof|3dof) "6dof"
80          >
81
```

```
 1  <?xml version="1.0"?>
 2  <!DOCTYPE configFile SYSTEM "../include/configFile.dtd">
 3  <configFile>
 4
 5  <description>
 6        This is the minimum configuration.
 7        It only defines the marker broker settings and the parameters for
 8        the default marker.
 9  </description>
10
11  <author>
12          Created (using hand editing) by:
13          Daniel F. Johnston
14          National Research Council of Canada
15          2002 Dec. 10
16          Modified 2003 Jan 30 to support multiple destinations
17  </author>
18
19  <settings
20     sendRate="150"
21     >
22     <destination
23        sendPort="10035"
24        />
25     <destination
26        sendPort="10036"
27        sendAddress="dummy.imti.nrc.ca"
28        />
29     <destination
30        sendPort="10037"
31        sendAddress="origin1.imti.nrc.ca"
32        />
33     <dofx
34        name="X Loc"
35        maxLimit="100.0"
36        minLimit="-100.0"
37        />
38     <dofy
39        name="Y Loc"
40        maxLimit="120.0"
41        minLimit="-120.0"
42        />
43     <dofz
44        name="Z Loc"
45        maxLimit="110.0"
46        minLimit="-110.0"
47        />
48     <dofh
49        name="Heading"
50        maxLimit="180.0"
51        minLimit="-180.0"
52        />
53     <dofp
54        name="Pitch"
55        maxLimit="100.0"
56        minLimit="-100.0"
57        />
58     <dofr
59        name="Roll"
60        maxLimit="90.0"
61        minLimit="-90.0"
62        />
63  </settings>
64
65  </configFile>
```

```
<?xml version="1.0"?>
<!DOCTYPE configFile SYSTEM "../include/configFile.dtd">
<!--
  Please do not delete or change this file. It is used by
  the documentation for the 'marker broker' software.
-->

<configFile>

<description>
      This is a very basic configuration for testing.
      It defines the full set of required markerbroker settings,
      and then adds one set of dummy network markers (test_net),
      a test set of serial line markers, and one real set of serial line
      markers (Ascension FOB).
</description>

<author>
        Created (using hand editing) by:
        Daniel F. Johnston
        National Research Council of Canada
        2002 Nov. 20
        Modified 2003 Jan 30 to support multiple destinations
</author>

<settings
   sendRate="80.0"
   >
   <destination
      sendPort="10035"
      />
   <destination
      sendPort="10036"
      sendAddress="dummy.imti.nrc.ca"
      />
   <destination
      sendPort="10037"
      sendAddress="origin1.imti.nrc.ca"
      />
   <dofx
      name="Loc X"
      maxLimit="102.5"
      minLimit="-86.45"
      />
   <dofy
      name="Loc Y"
      maxLimit="120.0"
      minLimit="-97.3"
      />
   <dofz
      name="Loc Z"
      maxLimit="200.0"
      minLimit="-150.0"
      />
   <dofh
      name="Orient H"
      maxLimit="110.0"
      minLimit="-110.0"
      />
   <dofp
      name="Orient P"
      maxLimit="45.0"
      minLimit="-45.0"
      />
   <dofr
      name="Orient R"
```

```
67          maxLimit="90.0"
68          minLimit="-90.0"
69          />
70    </settings>
71
72    <network
73       readPort="10032"
74       numbMarkers="23"
75       type="test_net"
76       />
77
78    <serial
79       line="/dev/ttyd3"
80       baud="9600"
81       readRate="150.0"
82       numbMarkers="3"
83       type="test_serial"
84       />
85
86    <serial
87       line="/dev/ttyd2"
88       baud="19200"
89       readRate="150.0"
90       numbMarkers="1"
91       type="fob_ea"
92       />
93
94    </configFile>
```

```
<?xml version="1.0"?>
<!DOCTYPE configFile SYSTEM "../include/configFile.dtd">
<configFile>

<description>
      This is a very basic configuration for testing.
      It defines the usual full required set of markerbroker settings,
      and then adds one set of network markers (Motion Analysis TRC).
</description>

<author>
        Created (using hand editing) by:
        Daniel F. Johnston
        National Research Council of Canada
        2002 Nov. 20
</author>

<settings
   sendRate="1500.0"
   >
   <destination
      sendPort="10035"
      sendAddress="localhost"
      />
   <dofx
      name="Offset X"
      maxLimit="400.0"
      minLimit="-400.0"
   />
   <dofy
      name="Offset Y"
      maxLimit="400.0"
      minLimit="-400.0"
      />
   <dofz
      name="Offset Z"
      maxLimit="200.0"
      minLimit="-200.0"
      />
   <dofh
      name="Offset H"
      maxLimit="90.0"
      minLimit="-90.0"
      />
   <dofp
      name="Offset P"
      maxLimit="90.0"
      minLimit="-90.0"
      />
   <dofr
      name="Offset R"
      maxLimit="90.0"
      minLimit="-90.0"
      />
</settings>

<network
   readPort="10032"
   readAddress="pc0106-0227r.imti.nrc.ca"
   numbMarkers="11"
   scale="0.03937"
   type="ma_trc3d"
   />

</configFile>
```

Appendix B

MarkerBroker Connection Applications
Source code listings

Source files included in this appendix:

1. test_net.C: A test program that attaches to the shared memory, simulates a connection to a network device, and generates a set of fixed or random numbers that are stored in the shared memory to simulate position markers.

2. ma_htr2.C: This program will connect via the computer network to a Motion Analysis Corp. tracking system and read joint position and angle data. The data format from the tracking system will be the vendor's 'htr2' format.

3. ma_trc3d.C: This program will connect via the computer network to a Motion Analysis Corp. tracking system and marker data as 3D position (no orientation). This raw data format from the tracking system is the vendor's 'trc' format. An orientation value of (0,0,0) is assigned to each tracked marker.

4. test_serial.C: A test program that attaches to the shared memory, simulates a connection to a serial device, and generates a set of fixed or random numbers that are stored in the shared memory to simulate position markers.

5. fob_ea.C: This program will communicate via a serial line connection to a Ascension "Flock of Birds" magnetic tracking system. It is assumed that the tracking hardware is configured for 'extended addressing' mode. Each sensor in a 'flock' will provide all six values for position and orientation, so these values are stored in the shared memory.

```
 1  //
 2  // A simple program to test the marker broker application by supplying
 3  //   a task to write values into the set of shared current markers.
 4  //
 5
 6  #include "../../../include/shmVector.h"
 7  #include <iostream> // for cout, cerr
 8  #include <sstream>  // for formatting with string streams
 9  #include <stdlib.h> // for rand
10  #include <unistd.h> // for sleep
11  #include <time.h>   // for random number seed (current time)
12  #include <signal.h> // for signals ...duh!
13
14  // set on when the process is to exit
15  int exitFlag;
16
17  /**
18      \ingroup mod_cnetconnect
19      This function will be called by the sub-process in response
20      to a 'stop' signal.  It will cause the process to 'stop' by
21      changing the value of a shared variable;
22  */
23  static void
24  thisProcessStop( int dummy )
25  {
26      exitFlag = 1;
27  //std::cerr << "Test net process stop " << dummy << std::endl;
28  }
29
30  main(int argc, char *argv[] )
31  {
32      // we want the child process to respond to a request to stop
33      signal( SIGTERM, thisProcessStop );
34      exitFlag = 0;
35
36      // this program must connect to the shared memory vector
37      shmVector<float> ourValues( "any name" );
38
39      // the command line arguments should tell us where to
40      //  start writing and how many markers to write
41      if(argc != 5)
42          exit(-1);
43      int numberMarkers = atoi( argv[1] );
44      int startPosition = atoi( argv[2] );
45      float scaleToCommon;
46      sscanf(argv[3], "%f", &scaleToCommon );
47      int listenPort = atoi( argv[4] );
48  //std::cerr << "Test net " << argv[0] << " " << numberMarkers <<
49  //      " " << startPosition << " " << startPosition <<
50  //      " " << listenPort << std::endl;
51      // our numbers of markers is n, which is is 6*n floats
52      numberMarkers = numberMarkers*6;
53      // our start position is calculated in markers, there are 6 floats per
54      startPosition = startPosition*6;
55
56      // seed the random number generator
57      srand( (int)time(0) );
58      // now loop forever, creating random vector values
59      do
60      {
61          for( int i = 0; i < numberMarkers; i++ )
62          {
63              ourValues[startPosition+i] = (float)(i+1)*10.0f * scaleToCommon;
64  //          std::cerr << startPosition+i << " "
65  //                    << ourValues[startPosition+i] << std::endl;
66          }
```



```
67          sleep(2);
68      } while( exitFlag == 0 );
69
70      // we can stop, delete access to the shared memory vector, close
71      //  all sockets, etc.
72      return 0;
73  }
```

```
//
// A simple program to feed marker broker application by supplying
//   values into the set of shared current markers from a Motion
//   Analysis system and HTR2 format data.
//

#include "../../../include/shmVector.h"
#include <string>
#include <iostream> // for cout, cerr
#include <sstream>  // for formatting with string streams
#include <signal.h> // for signals ...duh!
#include <vector>
#include <unistd.h> // for _exit
#include <sys/types.h> // for wait
#include <sys/wait.h>
#include "evart.h"  // for motion analysis library calls
using std::cout;
using std::cerr;
using std::endl;

/**
    \ingroup mod_cnetconnect
    This function will be called by the sub-process in response
    to a 'stop' signal.  It will cause the process to 'stop' by
    changing the value of a shared variable;
*/
static void
thisProcessStop( int dummy )
{
//cerr << "Ma_htr2 net process stop " << dummy << endl;
}

/// an arbitary value for segments in case one not supplied
#define NUMSEG  4

/**
    The Motion Analysis SDK does not allow any user defined parameters
     to be passed to the data handler, so we must create a global
     (i.e. global to this source file) variable to allow the data
     handler to write data to the stared memory.
*/
shmVector<float> ourValues( "any name" );
/// To write to shared memory we need to know how many to use
int globalNumbMarkers;
/// To write to shared memory we need to know where to start
int globalStartPosition;
/// we need to know the scale conversion factor
float globalScale;


/*=======================================================
//
// Note: MyDataHandler is called by the EVaRT thread.
//       That is not the main thread of the program.
//       Some operations cannot be performed across
//       threads.
//
//-------------------------------------------------------*/
int writeData(int iType, void *data)
{
    static int activeSegments = 0;
    static bool firstData = true;    // while True, we are waiting for data

    if (iType == TRC_DATA)
    {
        sTrcFrame *TrcFrame = (sTrcFrame *)data;
```



```

        cout << "TRC " << TrcFrame->iFrame << " -->" << endl;
    }

    else if (iType == GTR_DATA)
    {
        sGtrFrame *GtrFrame = (sGtrFrame *)data;

        cout << "GTR " << GtrFrame->iFrame << " -->" << endl;
    }

    else if (iType == HTR2_DATA)
    {
        sHtr2Frame *Htr2Frame = (sHtr2Frame *)data;

        // if we have not received the hierarachy message, then ignore data
        if( activeSegments == 0 )
            {
            cerr << "No hierarachy yet!" << endl;
            return 0; // I do not know what will happer if I return anything
                      //  but zero here.  Will the SDK die?
            }

        // if first data, then make an array of zero 'last values'
        if( firstData == true )
            {
            // check that the number of segments and expected markers match
            //  If more than expected, then only send the expected
            if( activeSegments*6 > globalNumbMarkers )
                activeSegments = globalNumbMarkers/6;
            //  if less than expected, send zero 'last message' data for extra

            // what follows will be our first real data message
            firstData = false;
            }

        // write the data to shared memory (if not 'out of range')
        int outOfRange = XEMPTY - 1.0f;  // so we can do 'greater than' rather
                                         //  than 'equal to' a floating number
        int i, k;
        for( i = 0, k = 0; i < globalNumbMarkers; i+=6, k++ )
        {
            // we only save new data if not out of range, else leave last
            if( Htr2Frame->Segments[k][0] < outOfRange )
                ourValues[globalStartPosition+i+0] =
                        Htr2Frame->Segments[k][0] * globalScale;
//          cout << "M[" << i/6 << "][0] = " <<
//                   Htr2Frame->Segments[k][0] << endl;
          if( Htr2Frame->Segments[k][1] < outOfRange )
              ourValues[globalStartPosition+i+1] =
                      Htr2Frame->Segments[k][1] * globalScale;
//          cout << "M[" << i/6 << "][1] = " <<
//                   Htr2Frame->Segments[k][1] << endl;
          if( Htr2Frame->Segments[k][2] < outOfRange )
              ourValues[globalStartPosition+i+2] =
                      Htr2Frame->Segments[k][2] * globalScale;
//          cout << "M[" << i/6 << "][2] = " <<
//                   Htr2Frame->Segments[k][2] << endl;
            // heading is a rotation around the z axis
          if( Htr2Frame->Segments[k][3] < outOfRange )
              ourValues[globalStartPosition+i+3] = Htr2Frame->Segments[k][3];
//          cout << "M[" << i/6 << "][3] = " <<
//                   Htr2Frame->Segments[k][3] << endl;
            // pitch is a rotation around the y axis
          if( Htr2Frame->Segments[k][4] < outOfRange )
              ourValues[globalStartPosition+i+4] = Htr2Frame->Segments[k][4];
```

```
133 //          cout << "M[" << i/6 << "][4] = " <<
134 //                      Htr2Frame->Segments[k][4] << endl;
135             // roll is a rotation around the x axis
136           if( Htr2Frame->Segments[k][5] < outOfRange )
137              ourValues[globalStartPosition+i+5] = Htr2Frame->Segments[k][5];
138 //          cout << "M[" << i/6 << "][5] = " <<
139 //                      Htr2Frame->Segments[k][5] << endl;
140 //          cout << "M[" << i/6 << "][6] = " <<
141 //                      Htr2Frame->Segments[k][6] << endl;
142 //          cout << endl;
143         }
144
145     }
146
147     else if (iType == HTR_DATA)
148     {
149         sHtrFrame *HtrFrame = (sHtrFrame *)data;
150
151         cout << "HTR " << HtrFrame->iFrame << " -->" << endl;
152     }
153
154     else if (iType == HIERARCHY)
155     {
156         sHierarchy *Hierarchy = (sHierarchy *)data;
157         int iSegment;
158         int nSegments = Hierarchy->nSegments;
159
160         activeSegments = nSegments;
161         cout << nSegments << " segments" << endl;
162         cout << "---------------" << endl;
163
164         for (iSegment=0 ; iSegment<nSegments ; iSegment++)
165         {
166             cout << "seg " << iSegment <<
167                 ": " << Hierarchy->szSegmentNames[iSegment] <<
168                 ", parent = " << Hierarchy->iParents[iSegment] << endl;
169         }
170         cout << endl;
171         cout << "--> " << endl;
172     }
173
174     else if (iType == MARKER_LIST)
175     {
176         sMarkerList *MarkerList = (sMarkerList *)data;
177         int iMarker;
178         int nMarkers = MarkerList->nMarkers;
179
180         cout << endl;
181         cout << nMarkers << " markers" << endl;
182         cout << "---------------" << endl;
183
184         for (iMarker=0 ; iMarker<nMarkers ; iMarker++)
185         {
186             cout << iMarker << ": " <<
187                 MarkerList->szMarkerNames[iMarker] << endl;
188         }
189         cout << endl;
190         cout << "--> " << endl;
191     }
192
193     else if (iType == MESSAGE)
194     {
195         cout << (char *)data << endl;
196         cout << "--> " << endl;
197     }
198
```



```
199     else if (iType == DISCONNECTED)
200     {
201         cout << "Disconnected" << endl;
202         cout << "--> " << endl;
203     }
204
205     return 0;
206 }
207
208 main(int argc, char *argv[] )
209 {
210     // we want the child process to respond to a request to stop
211     signal( SIGTERM, thisProcessStop );
212
213     // the command line arguments should tell us where to
214     //  start writing and how many markers to write
215     if(argc != 6)
216         exit(-1);
217     int numberMarkers = atoi( argv[1] );
218     int startPosition = atoi( argv[2] );
219     float scaleToCommon;
220     sscanf(argv[3], "%f", &scaleToCommon );
221     int readPort = atoi( argv[4] );
222     std::string listenAddress( argv[5] );
223 //cerr << "Ma_htr2 " << argv[0] << " " << numberMarkers <<
224 //      " " << startPosition << " " << scaleToCommon <<
225 //      " " << readPort << " " << listenAddress << endl;
226     // our numbers of markers is n, which is is 6*n floats
227     globalNumbMarkers = numberMarkers*6;
228     // our start position is calculated in markers, there are 6 floats per
229     globalStartPosition = startPosition*6;
230     // share our scale factor
231     globalScale = scaleToCommon;
232
233     // connect to the motion analysis system and set proper streaming mode
234     EVaRT_Initialize();
235     EVaRT_SetDataHandlerFunc(writeData);
236
237     if (!EVaRT_Connect(argv[5]))
238     {
239         cerr << "*** Unable to connect to EVaRT at " <<
240                         listenAddress << endl;
241         _exit(-1);
242     }
243
244     EVaRT_SetDataTypesWanted(HTR2_DATA);
245
246     // prime the data handler with the number of current markers
247     EVaRT_RequestHierarchy();
248
249     if (!EVaRT_IsStreaming())
250     {
251         EVaRT_StartStreaming();
252         /*EVaRT_UDP_StartStreaming();*/
253     }
254
255     // now pause this process, all work is done in the data handler
256     // Note: The following couple of lines does the equivalant of
257     //   a 'wait for signal' command, i.e. "wait( 0 );", but the
258     //   real wait command does not respond to all signals, so
259     //   we do this instead.
260     char line[256];
261     scanf("%s", line);
262
263     //  the program (if we get here) is now terminated, close the
264     //   connection to the tracking system
```

```
265     if (EVaRT_IsConnected())
266     {
267         EVaRT_Disconnect();
268     }
269
270     EVaRT_Exit();
271     // we can stop, delete access to the shared memory vector, close
272     //  all sockets, etc.
273     return 0;
274 }
```

```
//
// A simple program to feed marker broker application by supplying
//   values into the set of shared current markers from a Motion
//   Analysis system and simple (raw) trc format data.  The raw
//   data will be treated as 3 DOF data, and the rotations will be
//   set all as zero.
//

#include "../../../include/shmVector.h"
#include <string>
#include <iostream> // for cout, cerr
#include <sstream>  // for formatting with string streams
#include <signal.h> // for signals ...duh!
#include <unistd.h> // for _exit
#include <sys/types.h> // for wait
#include <sys/wait.h>
#include "evart.h"  // for motion analysis library calls

/**
    \ingroup mod_cnetconnect
    This function will be called by the sub-process in response
    to a 'stop' signal.  It will cause the process to 'stop' by
    changing the value of a shared variable;
*/
static void
thisProcessStop( int dummy )
{
//std::cerr << "Ma_trc3d net process stop " << dummy << std::endl;
}

/// an arbitary value for segments in case one not supplied
#define NUMSEG  4
/// an arbitary value for markers in case one not supplied
#define NUMMARKER  4
/// a value that indicates a bad data value
#define BAD_DATA XEMPTY-10.0f

/**
    The Motion Analysis SDK does not allow any user defined parameters
     to be passed to the data handler, so we must create a global
     (i.e. global to this source file) variable to allow the data
     handler to write data to the stared memory.
*/
shmVector<float> ourValues( "any name" );
/// To write to shared memory we need to know how many to use
int globalNumbMarkers;
/// To write to shared memory we need to know where to start
int globalStartPosition;
/// we need to know the scale conversion factor
float globalScale;

/*********************************************************
//
// Note: MyDataHandler is called by the EVaRT thread.
//       That is not the main thread of the program.
//       Some operations cannot be performed across
//       threads.
//
//--------------------------------------------------------*/
int writeData(int iType, void *data)
{
    static int activeSegments = 0;
    static int activeMarkers = 0;

    if (iType == TRC_DATA)
    {
```



```
        sTrcFrame *TrcFrame = (sTrcFrame *)data;

        if( activeMarkers == 0 )
            activeMarkers = NUMMARKER;

        // write the data to shared memory
        int i, k;
        for( i = 0, k = 0; i < globalNumbMarkers; i+=6, k++ )
        {
            // check for valid marker data
            if( TrcFrame->Markers[k][0] > BAD_DATA |
                    TrcFrame->Markers[k][1] > BAD_DATA |
                    TrcFrame->Markers[k][2] > BAD_DATA )
                continue;  // this marker is invalid, use previous value
            ourValues[globalStartPosition+i] =
                        TrcFrame->Markers[k][0] * globalScale;
            ourValues[globalStartPosition+i+1] =
                        TrcFrame->Markers[k][2] * globalScale;
            ourValues[globalStartPosition+i+2] =
                        TrcFrame->Markers[k][1] * globalScale;
            // all rotation will be 0.0f
            ourValues[globalStartPosition+i+3] = 0.0f;
            ourValues[globalStartPosition+i+4] = 0.0f;
            ourValues[globalStartPosition+i+5] = 0.0f;
        }
    }

    else if (iType == GTR_DATA)
    {
        sGtrFrame *GtrFrame = (sGtrFrame *)data;

        printf("\rGTR %5d --> ", GtrFrame->iFrame);
    }

    else if (iType == HTR2_DATA)
    {
        sHtr2Frame *Htr2Frame = (sHtr2Frame *)data;

        printf("\nHTR2 %5d --> ", Htr2Frame->iFrame);
    }

    else if (iType == HTR_DATA)
    {
        sHtrFrame *HtrFrame = (sHtrFrame *)data;

        printf("\nHTR %5d --> ", HtrFrame->iFrame);
    }

    else if (iType == HIERARCHY)
    {
        sHierarchy *Hierarchy = (sHierarchy *)data;
        int iSegment;
        int nSegments = Hierarchy->nSegments;

        activeSegments = nSegments;
        printf("%d segments\n", nSegments);
        printf("--------------\n");

        for (iSegment=0 ; iSegment<nSegments ; iSegment++)
        {
            printf("seg %d: %s, parent = %d\n",
                iSegment,
                Hierarchy->szSegmentNames[iSegment],
                Hierarchy->iParents[iSegment]);
        }
        printf("\n");
```

```
133         printf("--> ");
134     }
135
136     else if (iType == MARKER_LIST)
137     {
138         sMarkerList *MarkerList = (sMarkerList *)data;
139         int iMarker;
140         activeMarkers = MarkerList->nMarkers;
141
142         printf("\n");
143         printf("%d markers\n", activeMarkers);
144         printf("--------------\n");
145
146         for (iMarker=0 ; iMarker<activeMarkers ; iMarker++)
147         {
148             printf("%2d: %s\n",
149                 iMarker,
150                 MarkerList->szMarkerNames[iMarker]);
151         }
152         printf("\n");
153         printf("--> ");
154     }
155
156     else if (iType == MESSAGE)
157     {
158         printf("%s\n", (char *)data);
159         printf("--> ");
160     }
161
162     else if (iType == DISCONNECTED)
163     {
164         printf("Disconnected\n");
165         printf("--> ");
166     }
167
168     fflush(stdout);
169
170     return 0;
171
172 }
173
174 main(int argc, char *argv[] )
175 {
176     // we want the child process to respond to a request to stop
177     signal( SIGTERM, thisProcessStop );
178
179     // the command line arguments should tell us where to
180     //  start writing and how many markers to write
181     if(argc != 6)
182         exit(-1);
183     int numberMarkers = atoi( argv[1] );
184     int startPosition = atoi( argv[2] );
185     float scaleToCommon;
186     sscanf(argv[3], "%f", &scaleToCommon );
187     int readPort = atoi( argv[4] );
188     std::string listenAddress( argv[5] );
189 //std::cerr << "Ma_trc3d " << argv[0] << " " << numberMarkers <<
190 //      " " << startPosition << " " << scaleToCommon <<
191 //      " " << readPort << " " << listenAddress << std::endl;
192     // our numbers of markers is n, which is is 6*n floats
193     globalNumbMarkers = numberMarkers*6;
194     // our start position is calculated in markers, there are 6 floats per
195     globalStartPosition = startPosition*6;
196     // share our scale factor
197     globalScale = scaleToCommon;
198
```



```
199     // connect to the motion analysis system and set proper streaming mode
200     EVaRT_Initialize();
201     EVaRT_SetDataHandlerFunc(writeData);
202
203     if (!EVaRT_Connect(argv[5]))
204     {
205         std::cerr << "*** Unable to connect to EVaRT at " <<
206                      listenAddress << std::endl;
207         _exit(-1);
208     }
209
210     EVaRT_SetDataTypesWanted(TRC_DATA);
211
212     // prime the data handler with the number of current markers
213     EVaRT_RequestMarkerList();
214
215     if (!EVaRT_IsStreaming())
216     {
217         EVaRT_StartStreaming();
218         /*EVaRT_UDP_StartStreaming();*/
219     }
220
221     // now pause this process, all work is done in the data handler
222     // Note: The following couple of lines does the equivalant of
223     //   a 'wait for signal' command, i.e. "wait( 0 );", but the
224     //   real wait command does not respond to all signals, so
225     //   we do this instead.
226     char line[256];
227     scanf("%s", line);
228
229     //  the program (if we get here) is now terminated, close the
230     //   connection to the tracking system
231     if (EVaRT_IsConnected())
232     {
233         EVaRT_Disconnect();
234     }
235
236     EVaRT_Exit();
237     // we can stop, delete access to the shared memory vector, close
238     //  all sockets, etc.
239     return 0;
240 }
```

```
//
// A simple program to test the marker broker application by supplying
//   a task to write values into the set of shared current markers.
//

#include "../../../include/shmVector.h"
#include <iostream> // for cout, cerr
#include <sstream>  // for formatting with string streams
#include <string>   // for c++ strings - couldn't you tell?
#include <stdlib.h> // for rand
#include <unistd.h> // for sleep
#include <time.h>   // for randon number seed (current time)
#include <signal.h> // for signals ...duh!

// set on when the process is to exit
int exitFlag;

/**
    \ingroup mod_cnetconnect
    This function will be called by the sub-process in response
    to a 'stop' signal.  It will cause the process to 'stop' by
    changing the value of a shared variable;
*/
static void
thisProcessStop( int dummy )
{
    exitFlag = 1;
//std::cerr << "Test serial process stop " << dummy << std::endl;
}

main(int argc, char *argv[] )
{
    // we want the child process to respond to a request to stop
    signal( SIGTERM, thisProcessStop );
    exitFlag = 0;

    // this program must connect to the shared memory vector
    shmVector<float> ourValues( "any name" );

    // the command line arguments should tell us where to
    //  start writing and how many markers to write
    if(argc != 7)
        exit(-1);
    int numberMarkers = atoi( argv[1] );
    int startPosition = atoi( argv[2] );
    float scaleToCommon;
    sscanf(argv[3], "%f", &scaleToCommon );
    std::string listenLine( argv[4] );
    int listenBaud = atoi( argv[5] );
    int listenRate = atoi( argv[6] );
//std::cerr << "Test serial " << argv[0] << " " << numberMarkers <<
//      " " << startPosition << " " << scaleToCommon << " " << listenLine <<
//      " " << listenBaud << " " << listenRate << std::endl;
    // our numbers of markers is n, which is is 6*n floats
    numberMarkers = numberMarkers*6;
    // our start position is calculated in markers, there are 6 floats per
    startPosition = startPosition*6;

    // seed the random number generator
    srand( (int)time(0) );
    // now loop forever, creating random vector values
    do
    {
        for( int i = 0; i < numberMarkers; i++ )
        {
            ourValues[startPosition+i] = (float)(i+1)*-10.0f * scaleToCommon;
```



```
//          std::cerr << startPosition+i << " "
//                    << ourValues[startPosition+i] << std::endl;
        }
        sleep(1);
    } while( exitFlag == 0 );

    // we can stop, delete access to the shared memory vector, close
    //  all sockets, etc.
    return 0;
}
```

```
//
// A program to read data from a flock of birds tracking system and feed
//   data to the marker broker application. It will provide the data
//   by writing values into the set of shared current markers.
//
// Note: This code depends on a modified public domain library for
//   controlling and reading data from a 'flock of birds'.  This
//   library is called "FlockLib".  NRC staff modified this library
//   to provide much better tolerance for hardware errors, to provide
//   setup with a configuration file, and to add support for the newer
//   Motionstar version of tracking hardware. The NRC modifications
//   are also freely available.
//

#include "../../../include/shmVector.h"
#include <iostream> // for cout, cerr
#include <sstream>  // for formatting with string streams
#include <string>   // for c++ strings - couldn't you tell?
#include <signal.h> // for signals ...duh!
#include "Flock.hh" // for communication from flock of birds

// set on when the process is to exit
int exitFlag;

/**
    \ingroup mod_serialconnect
    This function will be called by the sub-process in response
    to a 'stop' signal.  It will cause the process to 'stop' by
    changing the value of a shared variable;
*/
static void
thisProcessStop( int dummy )
{
    exitFlag = 1;
std::cerr << "fob_ea serial process stop " << dummy << std::endl;
}

main(int argc, char *argv[] )
{
    // we want the child process to respond to a request to stop
    signal( SIGTERM, thisProcessStop );
    exitFlag = 0;

    // this program must connect to the shared memory vector
    shmVector<float> ourValues( "any name" );

    // the command line arguments should tell us where to
    //  start writing and how many markers to write
    if(argc != 7)
        exit(-1);
    int numberMarkers = atoi( argv[1] );
    int startPosition = atoi( argv[2] );
    float scaleToCommon;
    sscanf(argv[3], "%f", &scaleToCommon );
    std::string listenLine( argv[4] );
    int listenBaud = atoi( argv[5] );
    int listenRate = atoi( argv[6] );
//std::cerr << "fob serial " << argv[0] << " " << numberMarkers <<
//      " " << startPosition << " " << listenLine <<
//      " " << listenBaud << " " << listenRate << std::endl;
    // our numbers of markers is n, which is is 6*n floats
    numberMarkers = numberMarkers*6;
    // our start position is calculated in markers, there are 6 floats per
    startPosition = startPosition*6;

    // now use the local configuration file to read flock parameters
```



```
    Flock flock("local.flock");
    float *data = new float[flock.getDataSize()];

    flock.startReading();

    // now loop forever, writting data values to shared memory
    do
    {
        // read the flock data at config'd measure and report rate
        if (!flock.read(data))
        {
            std::cerr << "Flock read failed!" << std::endl;
            exitFlag = 1;
            continue;
        }

        // now copy this data to shared memory
        for( int i = 0, loop = flock.getDataSize(); i < numberMarkers; i++ )
        {
//            std::cerr << i << data[i] << std::endl;
            if( i > loop )
                ourValues[startPosition+i] = data[i-loop] * scaleToCommon;
            else
                ourValues[startPosition+i] = data[i] * scaleToCommon;
        }
    } while( exitFlag == 0 );

    // we can stop, delete access to the shared memory vector, close
    //  all sockets, etc.
    delete data;
    return 0;
}
```