Independent project: An Extension of the CIS libraries

Alvin Liem
Dr. Russell Taylor
Fall 2002

Table of Contents

Project Overview

The goal of this project was to improve the CIS libraries through the addition of two modules. The first portion of the project consisted of extending the libraries to include support for two new tracking devices in the cisTracker module. The second part was the design and implementation of an error and status message logging system.

The various tracking systems currently available use different interfaces, different designs, and even different methods for tracking. We will examine these trackers in more detail. However, these devices all have the same general purpose: to identify the position of objects in space. Therefore, development of a common interface for these systems is natural. Programs can then be developed without using a specific interface, and different tracking devices can be used by a program without making any major changes to the program itself.

The diversity of these tracking devices also presents another problem. When there are errors in the operation of a tracking system, the cause can be very difficult to diagnose. Different tracking systems can have different types of errors. Additionally, errors can occur in various places not directly related to the tracking device, making it very difficult to determine where an error might be occurring. Therefore, should errors occur, they should be reported in a uniform manner that is complete, customizable and easy to understand. There should also be a way to log status messages, if the user wishes. This project addresses these issues.

Part 1: cisTracker

1.1 Background

Tracking devices are important for work in Computer Integrated Surgery (CIS). A tracking device is an apparatus that gathers information about the location of objects in space and returns that information relative to a known reference point. There are many different types of trackers currently available. Many of the trackers use optical technology to track objects. This usually involves placing cameras at various locations in the room and placing either active markers such as light emitting diodes on the objects to be tracked, or passive tools such as shiny spheres on the objects to be tracked. While this is an effective technology, one drawback of this approach is that the cameras must have a clear line-of-sight to the objects being tracked. Another major technology used in tracking devices is magnetic technology and is the one we will work closely with here. Specifically, this project deals with two magnetic trackers.

1.1a Aurora

The first tracking device is the experimental Aurora by Northern Digital. The Aurora consists of a magnetic field generator and a control module.

Fig. 1 – The Aurora Field Generator

The control module connects to the host computer and to tools with coils embedded within them. It communicates to the host computer via a serial port connection. The control module also connects to the tools in the setup. As the tools move around in space,

they are affected by changes in the magnetic field produced by the field generator, and their positions can be determined. This technology has the advantage that a clear line-of-sight is not necessary. In fact, a catheter can be inserted into a patient, and the location of the tip can still be known. Depending on the specific tool used, the Aurora can track 5 or 6 degrees of freedom. Because the tools have coils embedded within them, rotations about this axis cannot be tracked, and therefore 5 degrees of freedom can be tracked. If two coils which are not aligned are used in a single tool, 6 degrees of freedom can be tracked. There are a few potential drawbacks to this technology, however. First, the system is susceptible to magnetic interference. If there are large metal objects near the field generator, errors can be introduced. Second, it is important to keep in mind the distance from the field generator. As this distance is increased, the magnetic field becomes weaker and weaker. Eventually the object may pass outside the optimal tracking volume and the accuracy of the Aurora will drop, or the object may not be able to be tracked at all. As the Aurora is still in the developmental stages, these problems may eventually be addressed.

1.1b Flock of Birds

The second tracking device is the Flock of Birds by Ascension Technology. The Flock of Birds is also a magnetic tracking device.

Fig. 2 – The Flock of Birds

Like the Aurora, the Flock of Birds can track one or more objects from a single transmitter. Each tool is called a bird. There can be any number from one to 126 birds in

a flock, but as more birds are added, the speed of position updates deteriorates, and it is impractical to have much more than 10 birds in a flock. Each bird consists of a control unit and a sensor. One bird is designated the master and connects to the host computer via a serial port. One or more slave units can then be daisy chained to the master bird, thus comprising the Flock of Birds. This daisy chain connection is referred to as the Fast Bird Bus (FBB). To communicate with the various slave devices, the communications must pass through the master bird over the Fast Bird Bus. Also like the Aurora, there is a limit to the extent of the magnetic field produced by the device. The optimal tracking volume for the Flock of Birds is + or – 4 feet. With an extended range transmitter, the tracking volume is increased to + or – 8 feet. The Flock of Birds generates a single pulsed magnetic field that is measured by all the tools. This allows the calculation of the position of each individual tool. The Flock of Birds can track a tool at rates of up to 100 measurements per second when the serial communications are not the bottleneck in the system.

1.2 Design and Implementation

The first step was to design the extension of the cisTracker Application Programming Interface (API) to include the two additional devices. There is an existing cisTracker API for other tracking systems such as the Optotrak, Polaris and Flashpoint. This API allows applications to use a common interface for communication with the various tracking devices, even though each device has its own set of commands. Because each of the tracking devices perform similar functions, it is logical to have this common interface. The main function that these trackers have in common is to compute the

positions of tools. Therefore the cisTracker API has functions such as GetStatus() to return the status of the tracker, GetTools(vector<cisTrackerTool*>& ToolsOut) to return the tools associated with the tracker, StartTracking() to begin tracking, StopTracking() to end tracking, and Refresh() to update the positions of all the tools. These are the functions that we needed to define for the new tracking devices.

1.2a Aurora

We started with the development of the interface with the Aurora. As mentioned previously, the Aurora communicates through a serial port interface. It has three main modes of communication over the serial port. The first two are text based. That is, the commands are issued as a string and the results are returned in a textual format. The third mode of communication is binary based. That is, the commands are packed into a binary buffer, and the results are returned in a binary format. The textual mode of communication was used for debugging purposes, but once the structure of the interface was in place, we switched to the binary mode. As can be expected, due to the speed of the serial port, the binary mode is several times faster than the text mode.

The main task in setting up the Aurora for tracking is in the device initialization and the tool initialization. During the initialization process, the Aurora assigns a port handle to each tool. This port handle is dynamically assigned from a pool of available handles. The existing cisTracker interface refers to tools based on an index number. Therefore, we used an STL map to map tool indices to port handles.

Once the device and tools are set up, tracking is a matter of sending commands to the Aurora and reading back the replies. Each communication is verified using a CRC value. There were some implementation complications in that the data passed to and from

the device often had to be converted between ASCII, hexadecimal, and decimal values. However, once the details were sorted out, the implementation was straightforward.

There is a cisTracker base class which provides the skeleton for all cisTrackers. Therefore, we simply created a cisTrackerAurora class which inherits its members from cisTracker. This made the design of cisTrackerAurora very simple. We only had to implement the functionality of cisTracker with the Aurora specific code.

As in the other tracker specific code already implemented, we overrode the initialization functions. There are specific commands that we needed to send the Aurora to prepare it for tracking. This is accomplished through the Initialize(), InitializeTools(), and ActivatePorts() functions which are invoked through the constructor of cisTrackerAurora. We begin by initializing the serial port and then sending a "RESET" command to the Aurora. It is then possible to send the "INIT" command which initializes the hardware and prepares it for tool initialization. During the tool initialization, we first get a list of the ports that need to be enabled using "PHSR". Then, for each port that needs to be enabled, we send the unit a "PINIT" command to initialize the port, and a "PENA" command to enable the port. Following the tool initialization, we are now ready to start tracking. The other main functions we wrote were the StartTracking(), StopTracking() and Refresh() methods. These are the main methods used to actually get the tracking data from the device. StartTracking() sends the "TSTART" command to the Aurora. This puts the device in the mode to actually acquire the position data. Refresh() sends the "BX" command to the Aurora to do a position update using the binary communication format. Naturally, StopTracking() switches the mode of the Aurora out of the tracking mode by sending the "TSTOP" command. Other than these

main functions, we have code to perform the CRC checks to ensure that data has not been corrupted in the transmission from the Aurora control unit to the host computer. Additionally, as mentioned before, there is a fair amount of code which converts the various data formats used by the Aurora into formats that we can use easily and back.

1.2b Flock of Birds

Like the Aurora, the Flock of Birds also communicates through a serial port interface. However, the communications protocol is much simpler than with the Aurora. Most of the commands are composed of one byte and there are fewer commands. The replies to commands are always returned in binary unlike the Aurora. This, of course means that the communications will be more efficient than with a text based protocol. The initialization process of the Flock of Birds is fairly simple. There is no device initialization to perform initially. Instead, this is accomplished by toggling the switch on each bird from “standby” to “fly”. This allows the hardware to initialize and begin communications. The next step is to initialize the tools. With the Flock of Birds, a tool corresponds to a bird. We must send commands to the master bird to determine what each bird’s ID is and how many birds are in the flock. Then, we simply address each bird individually over the FBB and set the update mode on each device.

Once the birds are set up, tracking is a matter of sending commands to each bird individually and reading back the replies. The replies from the flock are in a special format specified by Ascension Technology.

As in our Aurora code, we overrode the initialization functions. This is accomplished through the same functions as described above: Initialize(), InitializeTools(), and ActivatePorts() which are invoked through the constructor of

cisTrackerFlockOfBirds. Again, we initialize the serial port and then query the master bird using the "EXAMINE VALUE" command. This gives us information about the number of birds in the flock as well as what the ID numbers are for each bird. Once we have this information, we can send each bird the command to set the output mode. For our implementation, we use the "POSITION/QUATERNION" mode. This puts us in the mode to return position and quaternion data for each tool update. In this implementation, StartTracking() and StopTracking() wake up the flock and put the flock to sleep respectively. The Refresh() method sends Refresh(index) commands to each bird individually. The reply data is sent in a binary format. We perform a series of low level bit operations (and, or, and bit shifts) on the data as specified by Acension Technology. Additionally, some fields need to be scaled to the device range of values. This gives us the position data of each tool in the format we need.

1.2c Testing

Once these interfaces were implemented, the remaining task was to test and document the code. Testing for the Aurora was performed using interface code provided by Northern Digital. This code was used to measure some tool positions and the same tool positions were set up using cisTracker. The calculations matched up, and the code was accepted as being correct. Likewise, the code for the Flock of Birds was tested against the Winbird application provided by Ascension Technology. The position of the tool was kept constant and both applications were used to measure the position. The tool was then moved and the positions were again measured with both applications. The numbers matched once again, and the code was accepted as correct.

1.2d cisTracker File Formats

In keeping with the format of cisTracker, there are a number of configuration files required to use either cisTrackerAurora, or cisTrackerFlockOfBirds. These files set the parameters for the tracking system. The formats of the general configuration files are identical for the Aurora and the Flock of Birds.

AuroraConfig.txt, FlockOfBirdsConfig.txt – These files provide general information about the tracking device. The valid fields are:

TRACKER_NAME - The name of the device, example: Aurora, FlockOfBirds

TRACKER_TYPE - The device type; This MUST match the type as defined in the code. Valid examples are: aurora, flockofbirds

TRACKER_CONFIG_FILE - File containing tracker specific information. Example: Aurora.ini, FlockOfBirds.ini

NUM_TOOLS - Number of tools, Example: 2

TOOL 1 - Tool initialization file, if necessary. Please see cisTracker documentation for more information. Example: AuroraTool1.ini

Aurora.ini, FlockOfBirds.ini – These files provide specific information about the tracking device. The valid fields are:

BAUD_RATE – The serial port communication rate. Example: 9600

SERIAL_PORT – The serial port number. Example: 2

Part 2: Addition of logging system

2.1 Background

Using these tracking devices in applications can be a challenging matter. Often errors can occur in the execution of an application, and the source of the problem can be

difficult to diagnose. Not only is it difficult to determine where an error has occurred, but it is also difficult to determine what went wrong. Furthermore, there are situations where a debugging tool would not work. For example, if there were problems in the serial port communications in a program, a debugging tool might not be helpful because the execution of the program would halt, and the communications with the external device would stop, possibly leading to a timeout error. Another situation might be where the position of the tool causes some kind of error in the program. This position might be hard to reproduce. Therefore, we need to be able to run the program normally in real time, while still giving more information about the actual execution of the program. Here, we attempted to address this problem by creating a flexible, easy to use system to log errors and status messages.

The goal was to create a comprehensive error and status logging system to assist in debugging and provide detailed information about the execution of a program. The requirements we placed on the solution were that it had to be flexible, easy to use, and it had to add minimal overhead. In particular, it had to be easy to configure in terms of what exactly is logged and how it is logged. Additionally, this should be accomplished without changing the logging code and without recompilation of the logging code. This is important because the user should not have to change things in the logging code.

We wanted the module to have a central class register so that we are able to identify each of the classes in a particular program. We wanted to have a Level Of Detail (LOD) setting tied to this class identification. The purpose of this is so that we know at any point in time whether a message should be logged based on the class that sent the message and its LOD setting. Furthermore, we wanted to be able to send the output of the

messaging system to many different destinations. These requirements proved quite challenging to meet, as is evident in the complexity of the solution.

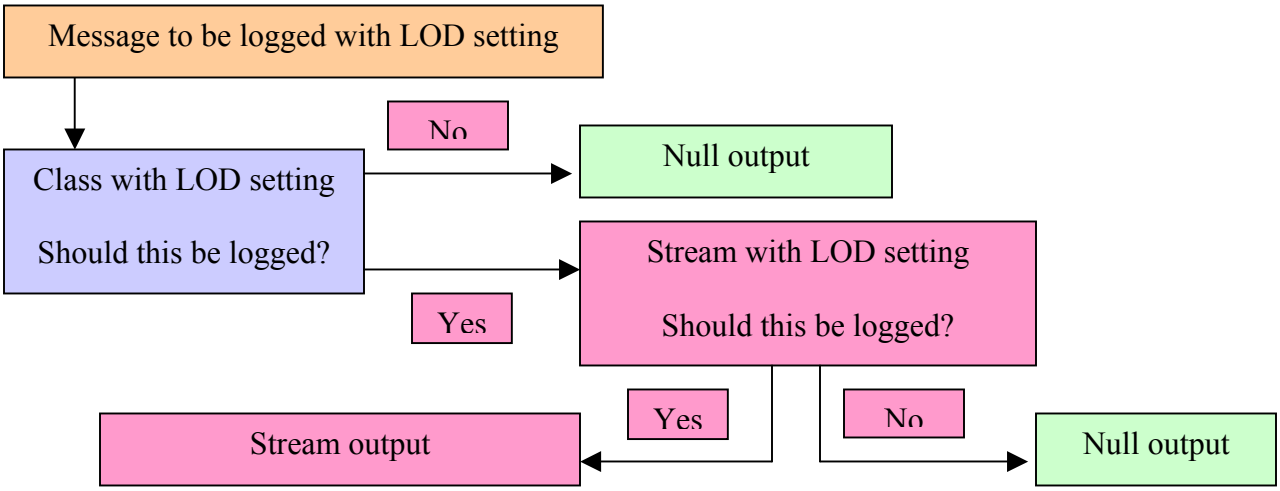


Fig. 3 – Layout of the messaging system

2.2 Design

The design of the solution might seem rather complicated at first glance, but it was necessary because of the constraints we placed on the solution.

2.2a cisClass

We started by defining a class called cisClass that represents a singleton identifier for a class. Therefore, for each class in any particular application, there would be one and only one cisClass object. This cisClass object also holds a text string representing the name of the class as well as the LOD setting for that class. This LOD setting is used to determine whether a message should be logged. In addition to giving us the ability to refer to a cisClass object by name, the string allows us to label the output with the class name. The name and LOD settings can be specified directly when the cisClass object is

defined, or it can be set through a system of user preferences which will be described later.

2.2b cisClassRegister

Each cisClass object is registered in the main class register. This design allows us to keep a single list of all the classes that exist in a particular program. The class register is implemented as a singleton so that it is unique and can be accessed from anywhere in a program. With this structure, we are able to access the class LOD setting in a central location, both through the actual class, as well as through the class name in the form of a string.

2.2c Stream Proxy and Multiplexer

The class register owns a stream proxy and stream multiplexer. There should only be one proxy and multiplexer for a given program, so this part is implemented as part of the class register. This allows us to meet the requirement of having multiple destinations as well as allowing us to control what information goes to which destinations. The proxy is used to determine if a message should be logged based on its LOD setting. The multiplexer allows the system to connect the output to multiple streams. We use streams as opposed to other methods for a number of reasons. Stream provide an easy way to log the messages because they are non-blocking. That is, the program logging the information will continue to execute while the information is being logged. This is important since we do not want to change the execution of a program simply because the amount of information being logged has changed. Also, there exists a lot of code in the CIS libraries which already use streams to output information, usually through the cout method to standard output. Additionally, since the "<<" operator is already overloaded

for different types of operands, we take advantage of existing code when we use streams. Furthermore, these streams can be of any type. This allows us to create new kinds of streams. For instance, we can log information to a file as well as to a status window. These streams have a LOD setting as well. This gives us the proxy functionality, thus giving us control over what information goes to which stream.

2.2d How it Works

There are a total of 3 LOD settings that come into play when determining whether a message should be logged. There is an LOD for the message itself. This defines the importance of the message itself. There is an LOD for the class through which the message is being logged. Finally, there is an LOD for each of the streams connected to the multiplexer in the class register. Therefore, there are two tests a message must pass in order to be logged. It must have a lower LOD setting than the value stored in the cisClass object, as well as a lower LOD setting than that stored in the stream. To understand how this mechanism works, consider a message which is sent to the class object. If the message's LOD is lower than the class LOD, the message is passed on to the class register. Otherwise, the message is destroyed and is not logged. If it is passed on, the class register's proxy/multiplexer unit then determines if the message should be logged. If the message's LOD is lower than the value stored with the stream, it is passed on to that stream. Otherwise it is sent to null output and is not logged. This design, while a bit complicated, allows us to have the functionality we initially required.

2.3 Implementation

2.3a Class Registration

Initially, we wanted to implement the class register as a simple static object. This would be done so that there would only be one class register in the program, and it would be accessible anywhere in the program. The idea is that each class then could register itself. This would minimize the amount of code required by developers of the new class. Registration would then be self contained and transparent to the developer. However, this caused some problems regarding the order of instantiation of objects as described below.

The macro cisREGISTER_CLASS adds a method called ClassInfo() to every class which contains the macro. This method simply returns a static data member of type cisClass. The method is used to ensure that the data member is instantiated and initialized the first time it is used. This is done to avoid problems related to the order of instantiation of the static data members.

For example, consider the situation where class A has a static cisClass object called classInfo. Assume that there is a class B which needs to use class A's classInfo member. If the code in A which contains the static definition of classInfo has not been executed yet, it will not exist and there will be a serious problem. This situation is frequently encountered when an object is defined as static outside of any code execution. To solve this problem, each time classInfo is needed, we simply create a method called ClassInfo() which returns the static data member. The data member will be created the first time the method is called, and will continue to exist until the program terminates.

The first time the ClassInfo() method is called, the cisClass object registers itself with the static class register. This is done through the constructor of the cisClass object.

Since the cisClass object is static, the constructor gets executed exactly once the first time the method is called.

2.3b Changing the LOD

The macro cisLOD("className", lod) sets the preference for the class named className to the LOD value specified in LOD. Alternatively, the LOD can be set directly using

```
myObject.ClassInfo()->SetLoD(lod); or
myClass::ClassInfo()->SetLoD(lod);
```

The register is implemented as a vector of pointers to the static cisClass objects that exist throughout the program. Initially, a map was used, but due to issues with Microsoft DLLs and STL maps, we chose to use a vector instead. A map would speed the string lookup of a cisClass object, and thus would be a better data structure to use, but unfortunately, we were not able to use it in this situation. It should be noted that due to the use of a vector, the cisClassRegister::GetClass() method performs a linear search on the vector and should be used sparingly. This method is used by the cisClassRegister::SetLoD("classname", LoD) method.

2.3c Preferences

The careful reader may have noticed that the cisClass object is instantiated and initialized the first time it is used. This includes when a message is logged, but what happens when the LOD is set but the ClassInfo() method has not been invoked and the cisClass object has not yet been instantiated? This situation can certainly happen, and the solution to the problem is to have class preferences. When the LOD is changed through one of the above methods, we do not do a blind update of the cisClass object. Instead, we

need to check if the object exists in the class register. If not, we create a class preference which saves the LOD setting before the cisClass object exists. Then, when the cisClass object is used, we use the LOD setting that was stored in the class preference. Through this solution, we have covered all of our bases, and there will not be any problems related to existence of the cisClass object.

At this point, each class has a static cisClass object (or a class preference), and there is a master list which contains all the pointers to those cisClass objects. It is now possible to add and remove channels to the stream multiplexer described above. Logging can then be used to output messages anywhere in the code.

2.3d Streams

Streams were initially a major roadblock due to the differences in stream implementation between Visual C++ and G++. These differences forced the upgrade to G++ 3.X. However, because G++ 3.X is still fairly new, there are still some problems in the stream code. This forced us to write a little stream hack to fix the problem temporarily.

When the logging code is used, a default output stream is created with the output file “cisLog.txt”.

To add channels to the output multiplexer, use code such as the following:

```
ofstream mainFile("mainOut.txt");
cisClassRegister::GetMultiplexer()->AddChannel(mainFile, 2);
cisClassRegister::GetMultiplexer()->AddChannel(cout,10);
```

The first two lines in this example add an output file called “mainOut.txt” to the multiplexer and sets the LOD setting for this file to 2. The last line simply adds the standard output stream to the multiplexer with an LOD setting of 10.

2.3e Logging

All of the macros used in logging can be used just like any other stream. That is, cisLOG(2)<< “hello” << endl; works great! The macro cisLOG(lod) simplifies the use of the Log method. However, it can only be used from within a class that has been registered using the cisREGISTER_CLASS method.

The macro cisLOG_DETAILED(lod) does the same thing as cisLOG, but provides the additional functionality of logging the file and line number at which the message was generated. As in the above macro, it can only be used from within a class that has been registered using the cisREGISTER_CLASS method.

The macro cisLOG_FUNC(lod) simplifies logging from outside a registered class. For example, it may be used inside the main() function. Unlike the two macros above, it does not provide any filtering based on the class level of detail, but it does provide filtering based on the output stream level of detail.

The macro cisLOG_FUNC_DETAILED(lod) is a macro to log from outside a registered class. Again, it does not provide any filtering based on the class level of detail. Similar to the cisLOG_DETAILED macro above, it provides information on the filename and line number of the log operation.

2.4 Notes on usage of the system

2.4a Basics

First, every class to be registered should contain the following bit of code in the public part of the class declaration:

```
cisREGISTER_CLASS(name, lod);
```

where name is a string representing the name to associate with that class, and lod is a cisShort representing the level of detail associated with that class. It is IMPORTANT to make sure that the name, provided as a string, exactly matches the class name.

Example:

```
class myClass {
        public:
        cisInt x, y;
        cisREGISTER_CLASS("myClass", 4);
}
```

This will register the class with the logging system and create a default error log in the form of a text file called “cisLog.txt.” From within that class, the macros described above can then be used. For example,

```
cisLOG(3) << "test error" << endl;
```

will simply send the message "test error" with the level of detail of 3 to the logging system.

To change the Level of Detail for a class, there are three possible ways:

- Via an implemented object: `myObject.ClassInfo()->SetLoD(lod)`.
- In the code, via the class: `myClass::ClassInfo()->SetLoD(lod)`.

- Anywhere with the class name as a string:

  cisClassRegister::SetLoD("myClass", lod).

One can also use the cisLOD macro to simplify this to cisLOD("myClass", lod). Once the class is registered with its own LOD and the output streams defined, each class can send its messages at different LODs.

2.4b Notes on Efficiency

It's important to understand that the message is first checked by the class itself before being passed to the streams. Therefore, removing the streams or setting a low LOD to all the streams is not the best way to reduce the cost of the logging. The best way is to set all the classes LOD at a low level. To do so, use the cisClassRegister::SetLoD(cisShort) which will perform an exhaustive replacement through all the known classes. To restore the default classes' LOD, one can use cisClassRegister::SetLoD(-1).

2.4c Recommended Levels of Detail

The specific levels of detail depend on each class' implementation. As we have described above, the LOD setting is specified on a class by class basis. Therefore, it is important that the messages in a class be consistent with each other in order to have useful message filtering based on the LOD settings. The LOD settings across classes do not have to be consistent, however it is better to be consistent since this would lead to increased usefulness of the stream levels of detail. Therefore, it is recommended to use some values between 0 and 10, 10 being the highest level of detail which corresponds to the maximum amount of logging. Also, for the programmers of new classes, we recommend the following levels:

- 1: Errors during the initialization.

- 2: Warnings during the initialization.

- 3 and 4: Extra messages during the initialization.

- 5: Errors during normal operations.

- 6: Warnings during normal operations.

- 7 and above: Very verbose.

The idea is that for most classes, important errors happen during the initialization (constructor, opening a cisSerialPort, creating a cisTracker, etc.) and during the normal operation, time can become critical. Therefore a level 5 would log a lot of information at the beginning and only the critical messages during the normal operations.

2.5 Logging Samples

The first example shows the normal operation of the Aurora with a high level of detail. As can be seen, there is a lot of data being written to the log:

```
cisClassRegister: The class preferences for cisNetworkPort have been
set with LoD 10 (class not yet registered)
cisClassRegister: The class preferences for cisSerialPort have been set
with LoD 10 (class not yet registered)
cisClassRegister: The class preferences for cisTracker have been set
with LoD 10 (class not yet registered)
cisClassRegister: The class preferences for cisTrackerPolaris have been
set with LoD 10 (class not yet registered)
cisClassRegister: The class preferences for cisTrackerAurora have been
set with LoD 10 (class not yet registered)
cisClassRegister: The class preferences for cisTrackerFlockOfBirds have
been set with LoD 10 (class not yet registered)
cisClassRegister: The class preferences for cisTrackerOptotrak have
been set with LoD 10 (class not yet registered)
cisClassRegister: The class preferences for cisTrackerSimulated have
been set with LoD 10 (class not yet registered)
cisClassRegister: The class preferences for cisTrackerFlashpoint have
been set with LoD 10 (class not yet registered)
cisClassRegister: The class preferences for cisTrackerPlayback have
been set with LoD 10 (class not yet registered)
cisClassRegister: The class preferences for cisTrackerFactory have been
set with LoD 10 (class not yet registered)
cisClassRegister: The class cisTrackerAurora has been registered with
LoD 10
cisTrackerAurora: Starting parsing of AuroraConfig.txt
```

```
cisTrackerAurora: Name of tracker: Aurora
cisTrackerAurora: Type of tracker: aurora
cisTrackerAurora: File for tracker configuration: Aurora.ini
cisTrackerAurora: Number of tools expected: 2
cisTrackerAurora: Ending parsing of AuroraConfig.txt
cisTrackerAurora: Load Configuration complete.
cisClassRegister: The class cisSerialPort has been registered with LoD
10
cisSerialPort: Set serial port speed to 9600 bauds
cisSerialPort: Sent a serial break and waited for 1 milliseconds
cisSerialPort: Received raw data  on 0 bytes
cisSerialPort: Received raw data  on 0 bytes
…
cisSerialPort: Received raw data  on 0 bytes
cisSerialPort: Received raw data RESETBE6 on 8 bytes
cisSerialPort: Received raw data F on 2 bytes
cisSerialPort: File: p:\liemat\cis-
2\source\cisnetwork\src\cisserialport.cpp Line: 550 IO Pending in Read
cisSerialPort: Sent raw data INIT:E3A5
 on 10 bytes
cisSerialPort: Received raw data OKAYA896
 on 9 bytes
cisSerialPort: File: p:\liemat\cis-
2\source\cisnetwork\src\cisserialport.cpp Line: 550 IO Pending in Read
cisSerialPort: Sent raw data COMM:500000048
 on 15 bytes
cisSerialPort: Received raw data OKAYA896
 on 9 bytes
cisSerialPort: Set serial port speed to 115200 bauds
cisTrackerAurora: Initialize complete.
cisTrackerAurora: Tools initialization.
cisTrackerAurora: File: p:\liemat\cis-
2\source\cistracker\src\cistrackeraurora.cpp Line: 326 using built in
SROM for tool
cisTrackerAurora: File: p:\liemat\cis-
2\source\cistracker\src\cistrackeraurora.cpp Line: 326 using built in
SROM for tool
cisTrackerAurora: InitializeTools complete.
cisSerialPort: File: p:\liemat\cis-
2\source\cisnetwork\src\cisserialport.cpp Line: 550 IO Pending in Read
cisSerialPort: Sent raw data PHSR:01E03E
 on 12 bytes
cisSerialPort: Received raw data 001414
 on 7 bytes
cisSerialPort: File: p:\liemat\cis-
2\source\cisnetwork\src\cisserialport.cpp Line: 550 IO Pending in Read
cisSerialPort: Sent raw data PHSR:02E17E
 on 12 bytes
cisSerialPort: Received raw data 020A00F0B00F6474
 on 17 bytes
cisSerialPort: File: p:\liemat\cis-
2\source\cisnetwork\src\cisserialport.cpp Line: 550 IO Pending in Read
cisSerialPort: Sent raw data PINIT:0AD5EB
 on 13 bytes
cisSerialPort: Received raw data  on 0 bytes
cisSerialPort: Received raw data  on 0 bytes
…
```

```
cisSerialPort: Received raw data  on 0 bytes
cisSerialPort: Received raw data OKAYA896
 on 9 bytes
cisSerialPort: File: p:\liemat\cis-
2\source\cisnetwork\src\cisserialport.cpp Line: 550 IO Pending in Read
cisSerialPort: Sent raw data PENA:0ADAD1E
 on 13 bytes
cisSerialPort: Received raw data  on 0 bytes
cisSerialPort: Received raw data OKAYA896
 on 9 bytes
cisSerialPort: File: p:\liemat\cis-
2\source\cisnetwork\src\cisserialport.cpp Line: 550 IO Pending in Read
cisSerialPort: Sent raw data PINIT:0BD4AB
 on 13 bytes
cisSerialPort: Received raw data  on 0 bytes
…
cisSerialPort: Received raw data  on 0 bytes
cisSerialPort: Received raw data OKAYA896
 on 9 bytes
cisSerialPort: File: p:\liemat\cis-
2\source\cisnetwork\src\cisserialport.cpp Line: 550 IO Pending in Read
cisSerialPort: Sent raw data PENA:0BD5D1E
 on 13 bytes
cisSerialPort: Received raw data  on 0 bytes
cisSerialPort: Received raw data OKAYA896
 on 9 bytes
cisTrackerAurora: Ports Initialized
Number of tools: 2
cisSerialPort: File: p:\liemat\cis-
2\source\cisnetwork\src\cisserialport.cpp Line: 550 IO Pending in Read
cisSerialPort: Sent raw data TSTART:5423
 on 12 bytes
cisSerialPort: Received raw data  on 0 bytes
…
cisSerialPort: Received raw data  on 0 bytes
cisSerialPort: Received raw data OKAYA896 on 8 bytes
cisSerialPort: Received raw data
 on 1 bytes
cisTrackerAurora: Start tracking at time 20589
cisTrackerAurora: USING BX!
cisSerialPort: File: p:\liemat\cis-
2\source\cisnetwork\src\cisserialport.cpp Line: 550 IO Pending in Read
cisSerialPort: Sent raw data BX:080100EC
 on 12 bytes
cisSerialPort: Received raw data Ä¥W on 95 bytes
All tools are visible
0 Tool #0: Position:[96.8547,-3.79587,17.5608] , : Orientation: [
0.773432 , [-0.0519179,0.480371,0.410307] ]
 (multiple points are tracked)
0 Tool #1: Position:[109.855,35.6995,47.9532] , : Orientation: [
0.392056 , [0.917967,0.0602347,0] ]
 (multiple points are tracked)
cisTrackerAurora: USING BX!
cisSerialPort: File: p:\liemat\cis-
2\source\cisnetwork\src\cisserialport.cpp Line: 550 IO Pending in Read
cisSerialPort: Sent raw data BX:080100EC
 on 12 bytes
```

```
cisSerialPort: Received raw data Ä¥W on 95 bytes
All tools are visible
1 Tool #0: Position:[96.8531,-3.79056,17.5532] , : Orientation: [
0.773355 , [-0.0518987,0.480425,0.41039] ]
 (multiple points are tracked)
1 Tool #1: Position:[109.854,35.6964,47.9538] , : Orientation: [
0.392052 , [0.917971,0.0602058,0] ]
 (multiple points are tracked)
…
```

The second example shows the normal operation of the Aurora with a lower LOD setting.

There is much less data being output to the log.

```
cisClassRegister: The class preferences for cisNetworkPort have been
set with LoD 2 (class not yet registered)
cisClassRegister: The class preferences for cisSerialPort have been set
with LoD 2 (class not yet registered)
cisClassRegister: The class preferences for cisTracker have been set
with LoD 3 (class not yet registered)
cisClassRegister: The class preferences for cisTrackerPolaris have been
set with LoD 10 (class not yet registered)
cisClassRegister: The class preferences for cisTrackerAurora have been
set with LoD 3 (class not yet registered)
cisClassRegister: The class preferences for cisTrackerFlockOfBirds have
been set with LoD 10 (class not yet registered)
cisClassRegister: The class preferences for cisTrackerOptotrak have
been set with LoD 10 (class not yet registered)
cisClassRegister: The class preferences for cisTrackerSimulated have
been set with LoD 10 (class not yet registered)
cisClassRegister: The class preferences for cisTrackerFlashpoint have
been set with LoD 10 (class not yet registered)
cisClassRegister: The class preferences for cisTrackerPlayback have
been set with LoD 10 (class not yet registered)
cisClassRegister: The class preferences for cisTrackerFactory have been
set with LoD 10 (class not yet registered)
cisClassRegister: The class cisTrackerAurora has been registered with
LoD 3
cisTrackerAurora: Starting parsing of AuroraConfig.txt
cisTrackerAurora: Name of tracker: Aurora
cisTrackerAurora: Type of tracker: aurora
cisTrackerAurora: File for tracker configuration: Aurora.ini
cisTrackerAurora: Number of tools expected: 2
cisTrackerAurora: Ending parsing of AuroraConfig.txt
cisTrackerAurora: Load Configuration complete.
cisClassRegister: The class cisSerialPort has been registered with LoD
2
cisTrackerAurora: Initialize complete.
cisTrackerAurora: Tools initialization.
cisTrackerAurora: File: p:\liemat\cis-
2\source\cistracker\src\cistrackeraurora.cpp Line: 326 using built in
SROM for tool
cisTrackerAurora: File: p:\liemat\cis-
2\source\cistracker\src\cistrackeraurora.cpp Line: 326 using built in
SROM for tool
```

```
cisTrackerAurora: InitializeTools complete.
cisTrackerAurora: Ports Initialized
Number of tools: 2
cisTrackerAurora: Start tracking at time 20489
All tools are visible
0 Tool #0: Position:[96.826,-3.78584,17.5328] , : Orientation: [ 0.7731
, [-0.0514552,0.480527,0.410807] ]
 (multiple points are tracked)
0 Tool #1: Position:[109.849,35.6938,47.9537] , : Orientation: [
0.392042 , [0.917976,0.0601868,0] ]
 (multiple points are tracked)
All tools are visible
1 Tool #0: Position:[96.8244,-3.78885,17.5344] , : Orientation: [
0.773123 , [-0.0514288,0.480507,0.41079] ]
 (multiple points are tracked)
1 Tool #1: Position:[109.847,35.6918,47.9536] , : Orientation: [
0.392039 , [0.917978,0.0601805,0] ]
 (multiple points are tracked)
All tools are visible
…
```

The third example shows a low level of detail with an error occurring. Notice the error regarding the CRC values:

```
cisClassRegister: The class preferences for cisNetworkPort have been
set with LoD 2 (class not yet registered)
cisClassRegister: The class preferences for cisSerialPort have been set
with LoD 2 (class not yet registered)
cisClassRegister: The class preferences for cisTracker have been set
with LoD 3 (class not yet registered)
cisClassRegister: The class preferences for cisTrackerPolaris have been
set with LoD 10 (class not yet registered)
cisClassRegister: The class preferences for cisTrackerAurora have been
set with LoD 3 (class not yet registered)
cisClassRegister: The class preferences for cisTrackerFlockOfBirds have
been set with LoD 10 (class not yet registered)
cisClassRegister: The class preferences for cisTrackerOptotrak have
been set with LoD 10 (class not yet registered)
cisClassRegister: The class preferences for cisTrackerSimulated have
been set with LoD 10 (class not yet registered)
cisClassRegister: The class preferences for cisTrackerFlashpoint have
been set with LoD 10 (class not yet registered)
cisClassRegister: The class preferences for cisTrackerPlayback have
been set with LoD 10 (class not yet registered)
cisClassRegister: The class preferences for cisTrackerFactory have been
set with LoD 10 (class not yet registered)
cisClassRegister: The class cisTrackerAurora has been registered with
LoD 3
cisTrackerAurora: Starting parsing of AuroraConfig.txt
cisTrackerAurora: Name of tracker: Aurora
cisTrackerAurora: Type of tracker: aurora
cisTrackerAurora: File for tracker configuration: Aurora.ini
cisTrackerAurora: Number of tools expected: 2
cisTrackerAurora: Ending parsing of AuroraConfig.txt
cisTrackerAurora: Load Configuration complete.
```

```
cisClassRegister: The class cisSerialPort has been registered with LoD
2
cisTrackerAurora: Initialize complete.
cisTrackerAurora: Tools initialization.
cisTrackerAurora: File: p:\liemat\cis-
2\source\cistracker\src\cistrackeraurora.cpp Line: 326 using built in
SROM for tool
cisTrackerAurora: File: p:\liemat\cis-
2\source\cistracker\src\cistrackeraurora.cpp Line: 326 using built in
SROM for tool
cisTrackerAurora: InitializeTools complete.
cisTrackerAurora: Ports Initialized
Number of tools: 2
cisTrackerAurora: Start tracking at time 21300
cisTrackerAurora: File: p:\liemat\cis-
2\source\cistracker\src\cistrackeraurora.cpp Line: 1002 Error - Body
CRC does not match
At least one tool is not visible
0 Tool #0: Not visible
0 Tool #1: Not visible
cisTrackerAurora: File: p:\liemat\cis-
2\source\cistracker\src\cistrackeraurora.cpp Line: 1002 Error - Body
CRC does not match
At least one tool is not visible
1 Tool #0: Not visible
1 Tool #1: Not visible
cisTrackerAurora: File: p:\liemat\cis-
2\source\cistracker\src\cistrackeraurora.cpp Line: 1002 Error - Body
CRC does not match
At least one tool is not visible
```

Part 3: Future Work

This project was designed to explore the various tracking devices and to help make them easier to use. We have extended the cisTracker infrastructure to include two new trackers, and we have created the cisLog system to help make software development using this system easier. Hopefully, others will find this software useful and helpful in the development of their own code. Future development along these lines might include extending the cisTracker implementation to include new future devices. According to Northern Digital, the Aurora and Polaris will eventually use the same communication interface. Therefore, future work may include combining the cisTrackerAurora interface with the cisTrackerPolaris interface to form a unified interface. Further development for

cisLog might include a graphical interface to display the classes registered and their associated LOD's. Such an interface might also include the ability to change the LOD of any class in a particular program.

What was learned

There were a great number of things that were learned through this project. In the first portion of the project, there were numerous problems in the development of the cisTrackerAurora and cisTrackerFlockOfBirds classes due to device communications issues. The communications were performed through the serial port of the host computer, and there were issues with the timing of commands and with the processing of the data that was returned following a command. It was learned that there are often a number of low level operations necessary to process the data returned from external devices such as the two tracking devices we worked with. Additionally, these operations may change at the manufacturer's whim, and this is precisely what occurred with the Aurora. A change in hardware occurred, and the Cyclic Redundancy Check (CRC) calculation routines changed. Thus, it is necessary to be able to adapt to changes, especially with experimental devices such as the Aurora.

In the second portion of the project, much was learned about what goes on "behind the scenes" with respect to the C++ programming language. When we wanted to implement the idea of a singleton, there were initial problems with how static objects are processed. Therefore, we had to implement them as methods which simply return static objects. Details of this solution are outlined above. Much knowledge was gained about how streams work, and how they are implemented using templates in the underlying code. In fact, we encountered problems with stream compatibility between Microsoft

Visual C++ and the GCC compilers. Additionally, we learned about the incompatibilities between the Standard Template Library (STL) and Microsoft Dynamic Linked Libraries (DLL's). Therefore, we had to establish workarounds for each of these problems. These workarounds involved upgrading our GCC compiler, as well as limiting the types of STL objects we used. We learned about the idea of class factories, stream proxies and stream multiplexers, as well as dynamic typing of objects in the code.

We also gained some good development skills. In particular, experience was gained in using tools such as CVS for version control in a group development environment. Additionally, the importance of using documentation tools such as Doxygen became apparent. The experience gained in using CVS and Doxygen alone makes the project worthwhile.

Finally, we learned about the importance of planning both in the design and implementation of the code for a project this size. We spent a considerable amount of time thinking about what was really required and what features would really be useful. It should be evident from the above that this project was very beneficial both for the developers as well as the users. Therefore, it appears that we have been quite successful in meeting our goals for this project.

Acknowledgements

Special thanks to Anton Deguet for his extensive support throughout this project. Thanks also to Ofri Sadowsky for his help with streams and the design of the logging system, and to Anand Viswanathan for the general support he provided.

Appendix

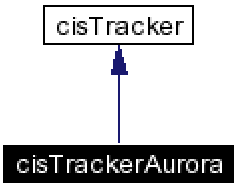
Following is the actual documentation of the cisTrackerAurora and cisClassRegister implementations as they exist in the code.

cisTrackerAurora Class Reference

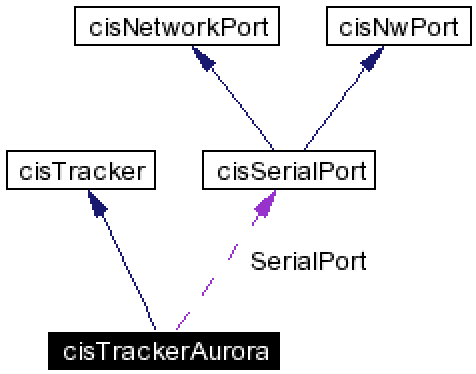
Aurora implementation for **cisTracker**.

`#include <cisTrackerAurora.h>`

Inheritance diagram for cisTrackerAurora:



Collaboration diagram for cisTrackerAurora:



Public Methods

- **cisREGISTER_CLASS** ("cisTrackerAurora", 5)
- **cisTrackerAurora** (char *configFile)
- **~cisTrackerAurora** ()
- **cisBool StartTracking** (void)
- **cisBool StopTracking** (void)
- **cisBool Refresh** (void)
- **cisBool Refresh** (const **cisUInt** index)
- **cisBool Reset** (void)
- **cisREGISTER_CLASS** ("**cisTracker**", 5)
- **cisTrackerStatus GetStatus** (void)
- **cisBool GetTool** (const **cisUInt** index, **cisTrackerTool** &ToolOut) const
- **cisBool GetTools** (vector< **cisTrackerTool** * > &ToolsOut)
- **cisInt GetNumberOfTools** (void)

Protected Methods

- **cisBool LoadConfiguration** (const **cisChar** *config_filename_in)
- **cisBool InitializeTools** (void)
- virtual **cisTrackerTool * InitializeTool** (const **cisChar** *ToolConfigFile)=0
- **cisBool CheckType** (void)

Protected Attributes

- **cisTrackerStatus Status**
- **cisUInt NumberOfTools**
- **cisChar * TrackerName**
- **cisChar * TrackerType**
- **cisChar * TrackerFile**
- vector< **cisTrackerTool** * > **Tools**
- vector< **cisChar** ** > **ToolFileList**

Private Methods

- **cisBool RefreshGX** (void)
- **cisBool RefreshBX** (void)
- char * **int2HexString** (int input)
- **cisBool HardwareReset** (void)
- **cisBool SendCommand** (**cisChar** *cmd, **cisChar** *retData, **cisInt** maxDataLength, **cisULong** delay=0)
- **cisBool GetResponse** (char *buffer, int maxLen)
- **cisBool SendBinaryCommand** (**cisChar** *cmd, **cisChar** *retData, **cisInt** maxDataLength, **cisULong** delay=0)
- **cisBool GetBinaryResponse** (char *buffer, int maxLen)
- **cisBool SetBaudRate** (unsigned long newBaudRate)
- unsigned **CalcCRC** (char *CRCData)
- void **InitCrcTable** ()
- unsigned int **CalcCrc16** (unsigned int crc, int data)
- unsigned int **SystemGetCRC** (char *string, int len)
- int **SystemCheckCRC** (char *string)
- float **GetFloat** (char *string)
- int **GetHex1** (char *string)
- int **GetHex2** (char *string)
- int **GetHex4** (char *string)
- **cisBool ExtractValue** (**cisChar** *str, **cisInt** len, **cisScalar** fDivisor, **cisScalar** &value)
- char * **ExtractData** (char *value, unsigned len)
- **cisTrackerTool * InitializeTool** (const char *configFile)
- **cisBool ActivatePorts** (void)
- **cisBool InitializePort** (int portNum)
- **cisBool EnablePort** (int portNum, char flag)
- **cisBool Initialize** (void)
- **cisChar * GetType** (void)
- bool **ReleasePortHandles** (void)

Private Attributes

- **cisSerialPort * SerialPort**
- int **currentTool**
- char **bxRetBuffer** [1024]
- bool **FirstCRC**

Detailed Description

The Aurora Tracker is a new Magnetic Tracker that uses only Active tools, and can track 5 degrees of freedom with single coil tools or 6 dof with two coil systems.

This class is used to communicate with the Aurora over a serial port connection. It supports text communication via the obsolete GX method as well as binary communication via the BX method. The BX method should be used at all times since it is much faster than the GX method.

Constructor & Destructor Documentation

cisTrackerAurora::cisTrackerAurora (char * *configFile*)

Constructor

Tracker specific constructor. The **LoadConfiguration**()method is called, then **Initialize**() (*p.36*) and **InitializeTools**() methods are called.

cisTrackerAurora::~cisTrackerAurora ()

Destructor

Deletes the SerialPort pointer **SerialPort** (*p.39*)

Member Function Documentation

cisBool cisTrackerAurora::RefreshGX (void) [private]

Performs a tool refresh operation on the Aurora

This method uses the GX protocol to perform the update. This means that the data returned from the Aurora is encoded in a textual format. For better performance, use the RefreshBX method.

Returns:

cisBool Signifies whether the operation was successful

cisBool cisTrackerAurora::RefreshBX (void) [private]

Performs a tool refresh operation on the Aurora

This method uses the BX protocol to perform the update. This means that the data returned from the Aurora is encoded in a binary format. This method gives better performance than the RefreshGX method.

Returns:

cisBool Signifies whether the operation was successful

char* cisTrackerAurora::int2HexString (int *input*) [private]

Converts an integer representation of a port handle to a hex string version.

The Aurora passes around tool handles as hex strings, but we represent them internally as integers; simply use this method to convert between the two.

Parameters:

input The integer represwnation of the port handle

Returns:

char* The hexadecimal string representation of the integer

cisBool cisTrackerAurora::HardwareReset (void) [private]

Completely resets the Aurora Hardware

The Aurora is reset by sending the console a serial break, after which Aurora is a clean slate (everything must be reinitialized).

Returns:

cisBool (true if OK)

cisBool cisTrackerAurora::SendCommand (cisChar * *cmd*, cisChar * *retData*, cisInt *maxDataLength*, cisULong *delay* = 0) [private]

Sends a command to Aurora through the serial port; also stores the reply data to a string reference

Parameters:

cmd command string sent to Aurora
retData reply string sent to host computer
maxDataLength maximum length of the reply string
delay for commands with delay, default is 0

Returns:

cisBool (true if OK)

cisBool cisTrackerAurora::GetResponse (char * *buffer*, int *maxLen*) [private]

Receives command from Aurora - submethod of SendCommand

Parameters:

buffer string into which Aurora reply is written
maxLen size of *buffer

Returns:

cisBool (true if OK)

cisBool cisTrackerAurora::SendBinaryCommand (cisChar * *cmd*, cisChar * *retData*, cisInt *maxDataLength*, cisULong *delay* = 0) [private]

Sends a binary command to Aurora through the serial port; also stores the reply data to a string reference

This method is used to send a BX (binary) command to the Aurora.

Parameters:

cmd command string sent to Aurora
retData reply string sent to host computer
maxDataLength maximum length of the reply string
delay for commands with delay, default is 0

Returns:

cisBool (true if OK)

cisBool cisTrackerAurora::GetBinaryResponse (char * *buffer*, int *maxLen*) [private]

Receives a binary command from Aurora - submethod of SendCommand

Parameters:

buffer string into which Aurora reply is written
maxLen size of *buffer

Returns:

cisBool (true if OK)

cisBool cisTrackerAurora::SetBaudRate (unsigned long *newBaudRate*) [private]

Sets baud rate between Aurora and host computer

Parameters:

newBaudRate baud rate

Returns:

cisBool (true if OK)

unsigned cisTrackerAurora::CalcCRC (char * *CRCData*) [private]

Calculates CRC value for a given string

Parameters:

CRCData string for which CRC is to be calculated

Returns:

unsigned CRC value

void cisTrackerAurora::InitCrcTable () [private]

Initializes the CRC table for calculating CRC's for binary communications

unsigned int cisTrackerAurora::CalcCrc16 (unsigned int *crc*, int *data*) [private]

Calculates a CRC16 (16 bit CRC) value for a given string

This routine calculates a running CRC16 using the polynomial X^16 + X^15 + X^2 + 1.

Parameters:

data data for which CRC is to be calculated

Returns:

unsigned CRC value

unsigned int cisTrackerAurora::SystemGetCRC (char * *string*, int *len*) [private]

Converts an integer representation of a port handle to a hex string version.

The Aurora passes around tool handles as hex strings, but we represent * them internally as integers; simply use this method to convert * between the two.

Parameters:

len The integer represwnation of the port handle

string * The hexadecimal string representation of the integer

int cisTrackerAurora::SystemCheckCRC (char * *string*) [private]

Converts an integer representation of a port handle to a hex string version.

The Aurora passes around tool handles as hex strings, but we represent * them internally as integers; simply use this method to convert * between the two.

Returns:

string* The hexadecimal string representation of the integer

float cisTrackerAurora::GetFloat (char * *string*) [private]

Converts a two-byte representation of a float to an actual float

Parameters:

string The byte of the float

Returns:

The float

int cisTrackerAurora::GetHex1 (char * *string*) [private]

Converts a byte hex representation of an integer to an actual int

Parameters:

string The byte of the int

Returns:

The int

int cisTrackerAurora::GetHex2 (char * *string*) [private]

Converts a two-byte hex representation of an integer to an actual int

Parameters:

string The two-byte int

Returns:

The int

int cisTrackerAurora::GetHex4 (char * *string*) [private]

Converts a four-byte hex representation of an integer to an actual int

Parameters:

string The four-byte int

Returns:

int The int

cisBool cisTrackerAurora::ExtractValue (cisChar * *str*, cisInt *len*, cisScalar *fDivisor*, cisScalar & *value*) [private]

Extracts a value from a Aurora reply segment

This method takes a string of numbers (preceded by a '+' or '-'), converts it into a signed number, then divides that number by a given divisor to obtain a resulting value.

Parameters:

str string containing value to be extracted
len length of the string 'vStr'
fDivisor number that extracted value is to be divided by
value variable in which resulting value is to be stored

Returns:

cisBool (true if OK)

char* cisTrackerAurora::ExtractData (char * *value*, unsigned *len*) [private]

Extracts a string value from an Aurora Reply Segment Used for returning the values form the port scan

Parameters:

value String containing the input
len Length of the input

cisTrackerTool* cisTrackerAurora::InitializeTool (const char * *configFile*) [private]

Creates a tracker tool based on a configuration file

Parameters:

configFile the tool configuration file

Returns:

cisTrackerTool * pointer to the created tool (NULL if error)

cisBool cisTrackerAurora::ActivatePorts (void) [private]

Activates the all the ports by scanning for handles then calling InitializePorts() and **EnablePort**() (*p.36*)

Returns:

cisBool true/false

cisBool cisTrackerAurora::InitializePort (int *portNum*) [private]

Initializes a port by passing PINIT

Parameters:

portNum The port number

Returns:

cisBool true/false

cisBool cisTrackerAurora::EnablePort (int *portNum*, char *flag*) [private]

Enables a porta with PENA

Parameters:

portNum the port number
flag

Returns:

cisBool true/false

cisBool cisTrackerAurora::Initialize (void) [private, virtual]

Initializes the Tracker - Sets tracker parameters

Tracker parameters are read from a tool config file:

- BAUD_RATE - desired Baud rate of the ser. connection (possible values: 9600, 14400, 19200, 38400, 57600, 115200)
- SERIAL_PORT - port number (possible values: 1, 2, 3, 4)

HardwareReset() (*p.33*) is then called (analogous to turning Aurora off, then on again), then the 'PINIT:' command is sent to the tracker; **SetBaudRate**() (*p.34*) is called afterwards (which sends commands to Aurora regarding communication parameters).

Returns:

cisBool (true if OK)

Reimplemented from **cisTracker**.

cisChar* cisTrackerAurora::GetType (void) [private, virtual]

Returns string containing tracker type name ("aurora")

Returns:

char*

Implements **cisTracker**.

bool cisTrackerAurora::ReleasePortHandles (void) [private]

Frees the ports by sending command PHF

Returns:

true/false

cisTrackerAurora::cisREGISTER_CLASS ("cisTrackerAurora", 5)

Class registration. Registers the class cisTrackerAurora with a default level of detail 5 for the logging.

The levels of detail are the same as the ones used for the class **cisTracker**.

See also:

cisClassRegister (*p.40*) **cisClass**

cisBool cisTrackerAurora::StartTracking (void) [virtual]

Starts tracking tool/marker motion

Sends the command 'TSTART:' to Aurora, causing it to begin tracking.

Returns:

true if OK

Reimplemented from **cisTracker**.

cisBool cisTrackerAurora::StopTracking (void) [virtual]

Stops tracking tool/marker motion

Sends the command 'TSTOP:' to Aurora causing it to stop tracking.

Returns:

true if OK

Reimplemented from **cisTracker**.

cisBool cisTrackerAurora::Refresh (void) [virtual]

Refreshes all tools' positional and orientation data.

Sends the command 'GX' to Aurora (along with a parameter value dependent on what is being tracked); a reply string containing the latest transformation info on what is being tracked is parsed and appropriately stored in the **Tools** vector.

Returns:

true if OK

Implements **cisTracker**.

cisBool cisTrackerAurora::Refresh (const cisUInt *index*) [virtual]

Performs the identical function as **Refresh**(void) (*p.37*)

This method is included for tracking systems which have an optimized method to refresh specific tools. The Aurora does not have such a function, so this method simply calls the above refresh method.

Parameters:

index vector index of desired tool

Returns:

true if OK

Implements **cisTracker**.

cisBool cisTrackerAurora::Reset (void) [virtual]

Resets Aurora

This method first completely resets Aurora through the ResetHardware(), then re-initializes the tracker and the tools; **StartTracking**() (*p.37*) can be called immediately after calling this method.

Returns:

true if OK

Reimplemented from **cisTracker**.

cisTracker::cisREGISTER_CLASS ("cisTracker", 5) [inherited]

Class registration. Registers the class **cisTracker** with a default level of detail 5 for the logging.

The following levels of detail have been used for all the classes of the **cisTracker** module (i.e. **cisTrackerTool**, cisTrackerAurora, **cisTrackerPolaris**, cisTrackerOptotrack, etc.):

- 1: Errors during the initialization.
- 2: Warnings during the initialization.
- 3: Extra information during the initialization.
- 5: Errors during the tracking.
- 6: Warnings during the tracking.
- 7: Time of refresh and tool status.
- 8: Positional information of tools displayed
- 9: Low level API and Port commands.

See also:

cisClassRegister (*p.40*) **cisClass**

cisTrackerStatus cisTracker::GetStatus (void) [inline, inherited]

Returns the status of the Tracker

Returns:

cisTrackerStatus object in which the current tracker status is written

cisBool cisTracker::GetTool (const cisUInt *index*, cisTrackerTool & *ToolOut*) const [inline, inherited]

Returns a specific **cisTrackerTool**

Parameters:

index vector index of desired tool
ToolOut reference to **cisTrackerTool** object to hold the results

Returns:

cisBool (true if OK)

cisBool cisTracker::GetTools (vector< cisTrackerTool * > & *ToolsOut*) [inline, inherited]

Returns the vector of current tools being tracked

Parameters:

ToolsOut vector of Tools that will be filled

Returns:

cisBool (true if OK)

cisInt cisTracker::GetNumberOfTools (void) [inline, inherited]

Returns the number of tools in the vector

Returns:

the number of tools in the vector **Tools**

cisBool cisTracker::LoadConfiguration (const cisChar * *config_filename_in*) [protected, inherited]

Loads overall configuration file, containing:

- a tracker name, tracker type, and the number of tools to be used
- a tracker hardware configuration file name

- tool configuration file names

Parameters:

config_filename_in the name of the configuration file

Returns:

true if OK

cisBool cisTracker::InitializeTools (void) [protected, inherited]

Initializes the Tools of the Tracker.

Returns:

cisBool (true if OK)

virtual cisTrackerTool* cisTracker::InitializeTool (const cisChar * *ToolConfigFile*) [protected, pure virtual, inherited]

Creates a tracker tool based on a configuration file. Implementation specific.

Parameters:

ToolConfigFile the tool configuration file

Returns:

cisTrackerTool * pointer to the created tool (NULL if error)

Implemented in **cisTrackerSimulated** .

cisBool cisTracker::CheckType (void) [protected, inherited]

Checks if tracker type specified in the configuration file matches the implementation class being used.

Returns:

cisBool (true if OK)

Member Data Documentation

cisSerialPort* cisTrackerAurora::SerialPort [private]

Pointer to the serial port

int cisTrackerAurora::currentTool [private]

Index to keep track of the current tool being initailized only

char cisTrackerAurora::bxRetBuffer[1024] [private]

Temporary buffer used to hold the return data of a BX call

bool cisTrackerAurora::FirstCRC [private]

Keep track of whether we have initalized the CRC table

cisTrackerStatus cisTracker::Status [protected, inherited]

Status value assigned to the Tracker.

See also:

cisTrackerStatus

cisUInt cisTracker::NumberOfTools [protected, inherited]

The number of tools that are being tracked or loaded from a configuration file.

cisChar* cisTracker::TrackerName [protected, inherited]

The name of the tracking device.

cisChar* cisTracker::TrackerType [protected, inherited]

The type of the Tracking device; optotrak,polaris,aurora etc.

cisChar* cisTracker::TrackerFile [protected, inherited]

The name of the configuration file for the tracking device.

vector<cisTrackerTool*> cisTracker::Tools [protected, inherited]

Vector containing **cisTrackerTool** objects.

vector<cisChar**> cisTracker::ToolFileList [protected, inherited]

Vector containing the list of names of configuration files

The documentation for this class was generated from the following file:

- **cisTrackerAurora.h**

cisClassRegister Class Reference

Main register for all classes.

`#include <cisClassRegister.h>`

Public Types

- typedef vector< **cisClass** * > **ClassVector**

Static Public Methods

- **cisShort GetLoD** (void)
- **cisShort SetLoD** (**cisShort** lod)
- bool **Register** (**cisClass** *classPtr)
- bool **SetLoD** (const string &name, const **cisShort** lod)
- **ClassVector * GetVector** (void)
- **cisLODOutputMultiplexer** & **GetLog** (const **cisShort** &lod)
- **cisLODMultiplexerStreambuf**< char > * **GetMultiplexer** (void)
- const **cisClass * GetClass** (const string &name)
- **cisLODOutputMultiplexer Log** (const **cisShort** &lod)

Static Private Methods

- **cisShort * GetLoDPtr** (void)
- **cisClass * GetClassPtr** (const string &name)
- **cisInt GetClassIndex** (const string &name)

Detailed Description

This is the main class that handles all of the class identification and registration. The description of the proper usage of the error logging system follows.

First, _every_ class to be registered should contain the following bit of code in the public part of the class declaration:

```
cisREGISTER_CLASS(name, lod);
```

where name is a string representing the name to associate with that class, and lod is a cisShort representing the level of detail associated with that class. It is IMPORTANT to make sure that the name, provided as a string, exactly matches the class name. Example:

```
  class myClass {  public:  cisInt x, y;  cisREGISTER_CLASS("myClass", 4);  }
```

This will register the class with the logging system and created a default error log in the form of a text file. From within that class, the macros described above can then be used. For example,

```
  cisLOG(3) << "test error" << endl;
```

will simply send the message "test error" with the level of detail of 3 to the logging system.

To add additional destinations for the logging information, calls of the following form must be used:

```
  ofstream mainFile("mainOut.txt");  cisClassRegister::GetMultiplexer() (p.43)-
>AddChannel(mainFile, 2);  cisClassRegister::GetMultiplexer() (p.43)->AddChannel(cout,
10);
```

The first line of the preceding example create an output file stream called mainFile, directed to the file mainOut.txt. The second line adds the file stream as a channel in the cisClassRegister's Multiplexerstreambuf. The third line simply adds cout (standard output to a console window) to the Multiplexerstreambuf. Actually, any type of output stream may be used in this way.

To change the Level of Detail for a class, there are three possible ways:

1. Via an implemented object: myObject.ClassInfo()->SetLoD(lod).
1. In the code, via the class: myClass::ClassInfo()->SetLoD(lod).
1. Anywhere with the class name as a string: **cisClassRegister::SetLoD** (*p.42*)("myClass", lod). One can also use the **cisLOG** to simplify this to cisLOG("myClass", lod).

Here are the details of the implementation. The macro **cisREGISTER_CLASS** adds a method called ClassInfo() to every class. This method returns a static data member of type **cisClass**. This **cisClass** object is a unique identifier for that class. While the reasons for writing a method which returns a static data member might seem confusing, they are actually quite simple. This method is used to ensure that the data member is initialized the first time it is used. This is done to avoid problems related to the order of instantiation of the static data members.

When the ClassInfo() method defined in the **cisREGISTER_CLASS** macro is called, the **cisClass** object registers itself with the static class register. This allows us to keep a single list of all the classes that exist in a particular program. The register is implemented as a vector of pointers to the static **cisClass** objects that exist throughout the program. It should be noted that the **cisClassRegister::GetClass**() (*p.43*) method performs a linear search on the vector and should be used sparingly.

At this point, each class has a static **cisClass** object, and there is a master list which contains all the pointers to those **cisClass** objects. It is now possible to add and remove channels as outlined above. Logging can then be used to output messages anywhere in the code.

Once the class is registered with its own LoD and the output streams defined, each class can send its messages at different LoDs. It's important to understand that the message is first checked by the class itself before being passed to the streams. Therefore, removing the streams or setting a low LoD to all the streams is not the best way to reduce the cost of the logging. The best way is to set all the classes LoD at a low level. To do so, use the **cisClassRegister::SetLoD**(cisShort) (*p.42*) which will perform an exhaustive replacement thru all the known classes. To restore the default classes' LoD, one can use **cisClassRegister::SetLoD** (*p.42*)(-1).

As for the signification of the different level of details, they depend on each class implementation. Nevertheless, it's recommended to use some values between 0 and 10, 10 being the highest level of detail which corresponds to the maximum amount of logging. Also, for the programmers of new classes, we recommend the following levels:

- 1: Errors during the initialization.
- 2: Warnings during the initialization.
- 3 and 4: Extra messages during the initialization.
- 5: Errors during normal operations.
- 6: Warnings during normal operations.
- 7 and above: Very verbose.

The idea is that for most classes, important errors happens during the initialization (constructor, opening a **cisSerialPort**, creating a **cisTracker**, etc.) and during the normal operations, time can become critical. Therefore a level 5 would log a lot of information at the beginning and only the critical messages during the normal operations.

Member Typedef Documentation

typedef vector<cisClass*> cisClassRegister::ClassVector

Simple typedefs to make the use of the STL vector easier

Member Function Documentation

cisShort cisClassRegister::GetLoD (void) [inline, static]

Get the global level of detail (aka register level of detail as apposed to the class level of detail). If the value is negative, it means that the global LoD is not set and is not used. Therefore each class relies on its own LoD to filter the output.

Returns:

The global level of detail.

cisShort cisClassRegister::SetLoD (cisShort *lod*) [static]

Set the global level of detail. This method goes thru all the classes and set their level of detail to a given lod. Setting the global lod to a negative value, means that the global lod must not prevail it allows to restore the classes LoDs using their own default LoD. If the parameter lod is nul or positive, it means that this lod will be used by all the classes. The main use of this method is to inhibit or reduce all output as early as possible in the cisClassRegister logging flow. The best example being to set this lod to zero and thus supress the output.

Parameters:

lod The new global/register LoD.

Returns:

The previous global LoD.

bool cisClassRegister::Register (cisClass * *classPtr*) [static]

The Register method registers a class pointer in the static register. It MUST NOT be used directly. It is used by the cisREGISTER macro.

Parameters:

classPtr The pointer to the **cisClass** object.

Returns:

bool True if successful, false if the class has not been registered (e.g. one can not register twice). This might happen if a programmer doesn't give the right string name for the class to be registered.

bool cisClassRegister::SetLoD (const string & *name*, const cisShort *lod*) [static]

The SetLoD method allows the user to specify the lod for a specific class. It checks to see if the class is registered. If so, it updates the ClassInfo object directly. Otherwise, it creates a temporary class info until the class is actually registered. Therefore, the register contains not only the registered classes but also the ones which might be registered later.

Parameters:

name The name of the class.

lod The level of detail.

Returns:

bool True if the class is registered.

ClassVector* cisClassRegister::GetVector (void) [static]

Returns the vector which acts as the static class register.

Returns:

The vector of **cisClass** pointers.

cisLODOutputMultiplexer& cisClassRegister::GetLog (const cisShort & *lod*) [static]

Returns the **cisLODOutputMultiplexer** with a level of detail set to lod. This corresponds to the MESSAGE lod, not the class level or stream level lod's.

Parameters:

lod The level of detail for this multiplexer.

Returns:

cisLODOutputMultiplexer & The output multiplexer.

cisLODMultiplexerStreambuf<char>* cisClassRegister::GetMultiplexer (void) [static]

Returns the **cisLODMultiplexerStreambuf** directly. This allows manipulation of the streambuffer for operations such as adding or deleting channels for the stream..

Returns:

cisLODMultiplexerStreambuf <char>* The Streambuffer.

const cisClass* cisClassRegister::GetClass (const string & *name*) [static]

Get a Class by name. Returns null if the class is not known.

Parameters:

name The name to look up.

Returns:

The pointer to the **cisClass** object corresponding to name, or null if not known.

cisLODOutputMultiplexer cisClassRegister::Log (const cisShort & *lod*) [static]

Log method. Simply return an output stream with the message LOD of lod.

Parameters:

lod The LOD to associate with this message.

Returns:

The multiplexer. Can be used just like any other output stream.

cisShort* cisClassRegister::GetLoDPtr (void) [static, private]

Get a pointer on the register/global LoD. This method is required since the global LoD is implemented as a static variable of this method. This guarantees that this static value is always implemented before it is used. The initial value is -1, aka not used.

Returns:

The address of the register LoD.

cisClass* cisClassRegister::GetClassPtr (const string & *name*) [static, private]

Get a Class by name. Returns null if the class is not known. This is the private, non-const version

Parameters:

name The name to look up.

Returns:

The pointer to the **cisClass** object corresponding to name, or null if not known.

cisInt cisClassRegister::GetClassIndex (const string & *name*) [static, private]

Get the index for a specific class. It can be used in combination with GetVector to find (and modify) a pointer on a **cisClass** object (e.g. (***GetVector**() (*p.43*))[GetClassIndex("myClassName")].

Returns:

The index in the vector of **cisClass** pointers. If the class is not in the vector, returns -1.

The documentation for this class was generated from the following file:

- **cisClassRegister.h**