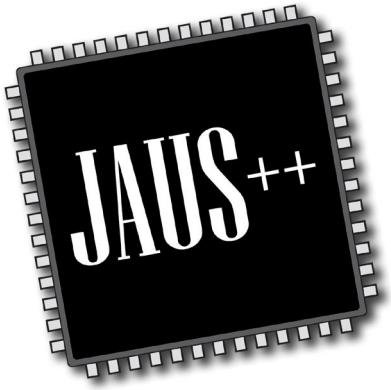


A C++ Implementation of JAUS

Version 1.0

ACTIVE Lab

JAUS++ Documentation

APPROVALS:

Final: Daniel Barber _____ Date: _____

Date	Reason for change	Authorization
7/30/07	Started first draft of document.	DB
9/1/07	Added additional information, including components and started describing the Node Manager interface.	DB
1/9/08	Updated documentation to reflect architecture/naming convention changes.	DB
2/29/08	Updated documentation to reflect 1.0 release.	DB
7/23/08	Updated components section, some cleanup and added logo	DB

1 Introduction

1.1 Joint Architecture for Unmanned Systems (JAUS)

JAUS, pronounced "jaws", is an emerging standard already in use on thousands of unmanned systems. It is a component-based message passing architecture that describes data fields and services. This standard is independent of current technology standards making it scalable for current and future hardware. This is achieved by only describing services and message passing rules while not requiring any additional implementation requirements.

The ACTIVE Laboratory has created a C++ based implementation of JAUS for use in real and simulated unmanned vehicles. The purpose of this document is to provide an introduction to the library for new developers, and give software architecture/implementation overview. After reading this document developers should know how to start using the JAUS++ library within an unmanned systems application. For more detailed information such as function parameters, class diagrams, etc. please refer to the JAUS++ Doxygen generated documents found with the library.

The ACTIVE Laboratory is part of the Institute for Simulation and Training located at the University of Central Florida with a focus on analyzing and improving human performance in virtual and mixed reality environments. More information can be found at http://active.ist.ucf.edu

1.2 Required Reading

Users are advised to look over the JAUS Reference Architecture (RA) documents first. They provide a more detailed overview (i.e. vocabulary, concepts, etc.). As of this writing the current RA version is 3.3. Parts 1 and 2 are necessary or users may have difficulty understanding vocabulary and interfaces described in this overview.

If you are already working on this project, or would like to contribute, read the ACTIVE Laboratory C++ Coding Standards document. It explains the coding styles and requirements used within this library. You can acquire these standards by contacting Daniel Barber – dbarber@ist.ucf.edu.

1.3 License

JAUS++ is distributed under the BSD License. The full description of this license is included with the library.

1.4 Common Vocabulary

The following section explains some common vocabulary used throughout this document.

I. JAUS RA – JAUS Reference Architecture or standards.
II. Component – A component is the lowest level of the system topology in JAUS. In this document a component is a software application that provides a specific service. Services provided can be GPS information, platform configuration data, or commands to other components, etc.

III. Node – A node has a collection of components running on it. A node is defined as any computing element within a subsystem that has a physical address (i.e. a serial port, IP address, etc.).

IV. Subsystem – A subsystem is a collection of nodes which represents an unmanned system. (i.e. a UGV is a subsystem, it has 4 nodes (computers) onboard, with each node containing different components (software applications))

V. System – A system is a collection of subsystems and is the highest level in the JAUS topology.

VI. Read/Write – Within the scope of this library reading or writing messages means to deserialize or serialize data. Serialization is the conversion of data in its native form to a byte array format as defined by the JAUS RA. Deserialization is the conversion of a JAUS formatted message within a byte array to a native/local data representation. It is recommended that those unfamiliar with the concept of serialization read more before continuing with this document.

1.5 JAUS++ Directory File Structure

- jaus++
  - include – Contains all header files for library
    - jaus – Main folder containing some common files
      - messages – Contains all message data structures
        - common – Contains common data structures used by message data structures
          - configuration – Dynamic configuration data structures
          - environment – Data structures for environment sensor subgroup
          - platform – Data structures for platform subgroup messages
          - manipulator – Data structures for manipulator subgroup messages
        - command – Folder for all command type messages (see commandcodes.h and commandmessages.h for lists of all command files and message codes)
          - core – All core subset JAUS messages
          - communications – Communications message subgroup
          - environment – Environment sensor subgroup
          - manipulator – Manipulator subgroup
          - platform – Platform subgroup
          - etc.
        - query – Folder for all query type messages (see querycodes.h and querymessages.h for lists of all query files and message codes)
          - Same layout format as command
        - inform – Folder for all inform type messages (see informcodes.h and informmessages.h for lists of all inform files and message codes)
          - Same layout format as command
        - experimental – Folder containing all experimental messages (custom messages).
      - components – Contains JAUS component interfaces ready to use
        - services – Ready to use components
          - node – Software for node manager component
        - transport – Files for handling transport layer

        - services – Contains interfaces for JAUS services (e.g. Global Pose Sensor)
        - video – Contains JAUS Video Streaming library files.
- src
    - Identical structure layout as include folder
- lib – Contains all compiled JAUS++ library files
- bin – Contains all compiled JAUS++ executable files, DLLs, and data needed for execution
- docs – Documentation
- build - Contains folders for compilation on different platforms
    - msvc8 – Microsoft Visual Studio 8 (2005) solutions
        - jaus++ – Builds main JAUS++ library
            - jmsglib
            - jcomponentslib
        - examples – Example programs
            - messsages
    - wm5 – Microsoft Visual Studio 8 Solutions for Windows Mobile 5 (structure similar to msvc8 folder)
    - linux – Contains Linux Makefiles for compilation
    - codeblocks – Contains workspaces and project files for Code::Blocks (Linux version)
- ext – Contains any external libraries needed to build JAUS++ library

1.6 JAUS++ Libraries and Applications

6.1.1 Message Library (jmsglib)

The Message Library contains all data structures describing JAUS messages. It has classes for the serialization and deserialization of data to formats defined in the JAUS Reference Architecture (RA).

6.2.1 Components Library (jcmblib)

The Components Library contains software for creating a Node Manager and Components. It implements all the functionality needed for sending and receiving messages.

6.3.1 Services Library (jsrvclib)

The Services Library defines interfaces for implementation of specific JAUS services described in Part 1 of the JAUS RA. Examples of services included are Global Pose Sensor, Velocity State Sensor, Primitive Driver, etc.

6.4.1 Video Library (jvideolib)

The Video Library contains an implementation of a Visual Sensor component capable of streaming image data in various formats. It does not include the means to get the image data (e.g. video capture), only the ability to compress RAW image data and stream it to subscribing components.

6.5.1 wxNodeManager

The wxNodeManager application is a JAUS Node Manager with a Graphical User Interface (GUI) that uses the wxWidgets library for cross-platform operation. See the Node Manager section of this document for more information.

6.6.1 wxVisualSensor

The wxVisualSensor application is Visual Sensor component capable of connecting to any DirectShow compatible video device or file in Windows. In Linux it can use any video source that is compatible with the OpenCV library from Intel.

6.7.1 wxVideoClient

The wxVideoClient application is a JAUS component application that can connect and subscribe to video data from a Visual Sensor Component. It is developed using wxWidgets for cross-platform operation.

1.7 Installation

All platforms that use this library must have the CxUtils library installed. If you do not have the distribution of JAUS++ that comes with CxUtils, you can download the latest version from our SourceForge project page (ACTIVE-IST) at: http://sourceforge.net/projects/active-ist/.

7.1.1 Visual Studio 2005

After installing CxUtils on the system, refer to the installation instructions in the CxUtils docs folder. CxUtils must be installed before setting up the JAUS++ library.

First, add the bin folder to your Windows PATH in the same way CxUtils bin was added. The DLL versions of the library are included with the SVN or other download formats.

Link to the libraries as needed depending on what features you need in your project. To build all the libraries, open up the jaus++.sln solution file in build/msvc8/jaus++ and perform a Batch Build of all the different configurations.

You do not need to configure your project to include the library files for JAUS++; they will be included automatically when you include the header files. If you plan to use the DLL versions of the library, you must use the appropriate pre-processor definitions.

To use the JAUS++ DLL files, you must also use the CxUtils DLL so make sure you add the CX_UTILS_DLL_IMPORTS and JAUS_DLL_IMPORTS preprocessor definitions to your project settings.

All example programs provided with the library compile using the DLL versions of CxUtils and JAUS++ libraries. Some examples need to load settings files from the bin directory, so if run from Visual Studio, do not forget to change the Working Directory to $(OutDir).

Linux

There are currently two options for building the JAUS++ libraries within a Linux environment. There are make files available and project/workspaces for the CodeLite IDE (http://codelite.org). To build using the Make, perform the following steps:

1. Install CxUtils (If already installed go to Step 2)
    a. From a terminal go to libraries/cxutils/trunk/1.0/build/linux
    b. Type make

c. Type sudo make install
d. To uninstall type sudo make uninstall

2. Install JAUS Libraries
    a. Build/Install Dependencies
        i. If not already installed install libjpeg
        ii. If not already installed install libpng
        iii. From a terminal go to ext/build/linux/tinyxml
            1. Type make
    b. From a terminal go to build/linux/jaus++ directory
    c. Type make
    d. Type sudo make install
    e. To uninstall type sudo make uninstall
3. Building Examples
    a. From a terminal go to build/linux/examples
    b. Type make
    c. Go to bin directory to run example programs
4. Building wxNodeManager
    a. Make sure you have the wxWidgets library installed
    b. From a terminal go to build/linux/wxapplications/wxnodemanager
    c. Type make
    d. Go to bin directory and type ./wxnodemanager to run

If you decide to use CodeLite, open the jaus++.workspace in build/codelite/jaus++ to build the library or you can open the build/codelite/examples/examples.workspace file which will build CxUtils, JAUS++, and JAUS++ Example Programs. Open the wxNodeManager.workspace in build/codelite/wxapplications/wxapplications.workspace to build the wxNodeManager application and any other applications that use wxWidgets. To build the wxvisualsensor program you will need to have OpenCV installed (http://sourceforge.net/projects/opencvlibrary/).

All executables will be placed in the bin directory.

1.8 Naming Conventions

The JAUS++ Library uses naming conventions defined by the ACTIVE Laboratory. This subsection provides a brief overview of the style used in the library.

8.1.1 Namespaces

Namespaces are written in PascalCase. Currently there is only one namespace in the JAUS++ library, and it is called “Jaus”.

8.2.1 Class Names

All classes and structures are written in PascalCase.
Examples:

```
Address
Header
Stream
LargeDataSet
```

```
SetGlobalWaypoint
```

8.3.1 Class Methods

All class methods are written in PascalCase. Arguments to methods are written using camelCase.

Examples:

```
int GetTimeMs() const;
int SetComponentAuthority(const Byte authorityCode);
Message* Clone() const;
Time UpdateTime();
```

8.4.1 Class Members

All class data members begin with the lowercase prefix ‘m’ for “member”, followed by a capital letter for the beginning of each new word in the variable name (camelCase). Any members that are also pointers begin with the prefix “mp”, for “member-pointer”. These are the only prefixes used in data member names.

Examples:

```
unsigned int mTimeStampMs;    ///<  Regular data member.
unsigned char *mpArrrayData; ///<  Pointer data members.
```

8.5.1 Functions/Methods

Similar to class names, functions/methods that are not part of a class use PascalCase.
Examples:

```
void ConvertRealToInteger(double, double, double, Int &)
void Sleep(const unsigned int ms)
```

8.6.1 Enumerations

All enumerations are written in PascalCase, including the enumeration name and variables.

Examples:

```
enum ResponseValues
{
      CreatedSuccessfully = 0,
      NodeDoesNotSupport,
      ComponentDoesNotSupport,
      Unused,
      Refused,
      InvalidParameters,
      MessageNotSupported,
};
```

8.7.1 Constants

All global constants are written in capital letters with words separated with an underscore, ‘_’, and all begin with the prefix “JAUS”.

Examples:

```
const UShort JAUS_HEADER_SIZE   = 16;   ///<  16 Bytes in JAUS Header.
const UShort JAUS_MAX_DATA_SIZE = 4095; ///< Max data field size.
```

2 Message Library

This section provides an overview of the JAUS++ message library. It describes the base types and classes for representing and interpreting JAUS messages. Whether or not the other features of the library are used for creation of components for communication, this core library is for message data storing and writing/reading to/from the byte array formats defined by the JAUS reference architecture.

2.1 Primitive Types

The JAUS architecture defines different primitive type values and how large each should be (byte, short integer, long integer, etc.). Therefore, several type definitions have been made to ensure the correct size variable is used to represent JAUS message data with different compilers/architectures. The following are the type definitions provided by JAUS++.

- Byte – Single unsigned byte
- UShort – 2 byte unsigned integer
- Short – 2 byte signed integer
- UInt – 4 byte unsigned integer
- Int – 4 byte signed integer
- ULong – 8 byte unsigned integer
- Long – 8 byte signed integer
- Float – 4 byte IEEE format floating point number
- LongFloat – 8 byte IEEE format floating point number

The reference architecture also describes the use of scaled integers to represent floating point number. To convert to and from Scaled Integers use the ScaledInteger class.

2.2 Address

The Address class is used to store JAUS ID information. All JAUS IDs are stored in an Address object. It contains methods to check if an ID is valid, broadcasting, etc. Any reference to a JAUS ID within the scope of this library is a reference to an Address structure.

2.3 Header

The Header class contains all JAUS message header information. This includes: Message Properties (JAUS version, ACK/NACK, etc.), Command Code (message type), Data Size, Data Control Flag, Sequence Number, Source ID, and Destination ID.

2.4 Stream

Besides Address and Header, Stream is the most commonly used class within this library. All serialized JAUS messages are stored in a Stream structure when sending or receiving. This is the version of a message that conforms to JAUS. It contains methods for reading (de-serialization) and writing (serialization) different data types into/from a byte array, handles memory allocation/deletion, and keeps track of the current reading/writing position in the byte array. Finally, if any byte order conversion is needed on a system, Stream handles it automatically. For example, all JAUS data is sent and received using Little-Endian byte order, so if your system

uses Big- Endian, the byte order is detected automatically and data read from the byte array is converted from Little-Endian automatically. When writing to a Stream on a Big-Endian system, the data is converted to Little-Endian also, automatically.

Message based classes (2.5), read and write data using a Stream. A Stream with a complete JAUS message includes header information, followed by the message body. **Do not read more than one message header and data block to a stream**. If the length of the byte array data in a Stream is greater than the maximum size allowed by JAUS (current limit is 4095 bytes), do not worry. Software within this library can be used to produce a Large Data Set (multi-packet stream) from an oversized Stream, or you can let the Component class interface (223.1) perform multi-packet stream handling automatically for you (recommended).

2.5 Message

To create a JAUS message structure, the Message class is used. **Message is a pure virtual class that all message structures inherit from.** It contains the minimum data required for a JAUS Message (command code, source and destination ID, etc.), functions to access and set these values, and six virtual functions that must be implemented by all classes that inherit from Message. Message provides a standard interface for reading/writing to/from a byte array, and for storing unserialized message data natively.

The functions that must be implemented are WriteMessageBody, ReadMessageBody, ClearMessageBody, Clone, GetPresenceVectorSize, and GetPresenceVectorMask. The WriteMessageBody and ReadMessageBody functions take a Stream object as their argument, and a version number. The WriteMessageBody function serializes the message body data members to a Stream following the specifications in JAUS and returns the number of bytes written to the stream (data size). The ReadMessageBody deserializes a message, saving the results to internal data members of the class and returning the number of bytes read. The ClearMessageBody method resets all data members except header information to their default values. The Clone function returns a pointer to a copy, (clone), of the Message that users must delete when finished. The GetPresenceVectorSize returns the size in bytes or the presence vector used by the message if one exists. If the message does not use a presence vector it returns a value of 0. The argument to this method is the JAUS version number in case the presence vector changes at a future date. Finally, the GetPresenceVectorMask method returns a bitmask which indicates what bits of the presence vector are used by the message. If no presence vector is used, then a value of 0 is returned. An example of a bitmask for a message that uses 5 bits would be 0x1F. This method also takes a JAUS version number as an argument.

All deserialization (reading from a Stream) can be done using the Read method of Message. All serialization (writing to a Stream) can be done using the Write method of Message. When writing, a Stream is cleared, a header is written, followed by message body data via the WriteMessageBody method. When reading, a Stream is not modified and its header is read, followed by the message body (using ReadMessageBody), and the results are saved to internal data members.

5.1.1 Example - How to Create a Message Class

A simple example to show how to create a JAUS message using the Message class is the Set Component Authority message. The name of the class should be identical to the message name, while following the naming conventions listed previously. Therefore, for this example the class name is SetComponentAuthority. It inherits from Message, and implements the required methods: WriteMessageBody, ReadMessageBody, ClearMessageBody, Clone, GetPresenceVectorSize, and GetPresenceVectorMask functions. It also has a copy constructor and overloaded `operator=` (all messages should do this), and has an additional data member which is a single byte that represents the authority value (from the JAUS standard). The following is the class definition.

```
class JAUS_EXPORT SetComponentAuthority : public Message
{
public:
    SetComponentAuthority();
    SetComponentAuthority(const SetComponentAuthority &msg);
    ~SetComponentAuthority();
    Byte GetAuthorityCode() const;
    int SetAuthorityCode(const Byte code);
    virtual int RunTestCase() const;
    virtual int WriteMessageBody(Stream &msg, const UShort version) const;
    virtual int ReadMessageBody(const Stream &msg, const UShort version);
    virtual void ClearMessageBody() { mAuthorityCode = 0; }
    virtual void Print() const;
    virtual Message* Clone() const;
    virtual UShort GetPresenceVectorSize(const UShort version =
                                                    JAUS_DEFAULT_VERSION) const;
    virtual UInt GetPresenceVectorMask(const UShort version =
                                                    JAUS_DEFAULT_VERSION) const;
    SetComponentAuthority &operator=(const SetComponentAuthority &msg);
protected:
    Byte mAuthorityCode; ///<  Authority code for command authority [0-255].
};
```

The Message class does not have a default constructor. It has one parameter that must be set which is the message type (Command Code). The following is the default constructor for SetComponentAuthority.

```
SetComponentAuthority::SetComponentAuthority() : Message(0x0001)
{
      mAuthorityCode = 0; // Default value.
}
```

The Message constructor must be initialized by your class constructor, so do not forget!

The Clone function returns a clone of the message data. The implementation is:

```
Message* SetComponentAuthority::Clone() const
{
      //  Allocate memory for message.
      SetComponentAuthority* ptr = new SetComponentAuthority();
      // Copies Message header data to message.
      CopyHeaderData(ptr);
      //  Copy the authority code data
```

```
        ptr->mAuthorityCode = this->AuthorityCode;
        // Return the pointer. It is up to the requestor to delete
        // the memory when done.
        return ptr;
}
```

For the above function, if a copy constructor is implemented and overloaded the assignment operator (recommended), then the above code can be reduced to:

```
Message* SetComponentAuthority::Clone() const
{
        return new SetComponentAuthority(*this);
}
```

The rest of the functions for this class should be self explanatory for those with programming experience, except for the WriteMessageBody and ReadMessageBody functions. Below are examples with full comments.

```
////////////////////////////////////////////////////////////////////////////
///
///   \brief Writes the message body data to the packet.
///
///   \param msg The packet to write the message body data to.
///   \param version The desired version of the message to write.
///
///   \return Number of bytes written on success.  A return of 0 is not
///           an error (some messages have no message body), only a
///           return of -1 and setting of an error code
///           is a failure state.
///
////////////////////////////////////////////////////////////////////////////
int SetComponentAuthority::WriteMessageBody(Stream& msg,
                                            const UShort version) const
{
    // Check the version of the message to write.
    // This way it is possible to add support for different
    // versions of JAUS.
    if(version <= JAUS_VERSION_3_3)
    {
        // Use the Write method of Stream to serialize
        // the data. This method returns the number of bytes
        // written.
        if(msg.Write( mAuthorityCode ))
        {
            // Return the number of bytes written to
            // the Stream instance (msg).
            return JAUS_BYTE_SIZE;
        }
    }
    else
    {
        //  Return -1 and set the specific
        //  error that took place when trying to write the
        //  message.  In this case, the version number set
```

```
        //  is not currently supported by this library.
        SetJausError(ErrorCodes::UnsupportedVersion);
        return -1;
    }

    // Return failure and signal that the function could not
    // write the data to the Stream by setting an Error flag.
    SetJausError(ErrorCodes::WriteFailure);
    return -1;
}

////////////////////////////////////////////////////////////////////////////
///
///   \brief Reads the message body data from the stream.
///
///   \param msg The stream to read from.
///   \param version The desired version of the message to read.
///
///   \return Number of bytes read on success.  A return of 0 is not
///           an error (some messages have no message body), only a
///           return of -1 and setting of an error code is
///           is a failure state.
///
////////////////////////////////////////////////////////////////////////////
int SetComponentAuthority::ReadMessageBody(const Stream& msg,
                                           const UShort version)
{
    // Check the version of the message to read.
    // This way it is possible to add support for different
    // versions of JAUS
    if(version <= JAUS_VERSION_3_3)
    {
        // Use the Read method of Stream to de-serialize
        // the data.  This method returns the number of bytes
        // read.
        if(msg.Read( mAuthorityCode ))
        {
            // Return the number of bytes read from
            // the Stream instance (msg).
            return JAUS_BYTE_SIZE;
        }
    }
    else
    {
        //  Return -1 and set the specific
        //  error that took place when trying to write the
        //  message. In this case, the version number set
        //  is not currently supported by this library.
        SetJausError(ErrorCodes::UnsupportedVersion);
        return -1;
    }

    // Return failure and signal that the function could not
    // write the data to the Stream by setting an Error flag.
    SetJausError(ErrorCodes::ReadFailure);
    return -1;
}
```

In the above code example, you'll notice the use of the SetJausError function whenever there is a failure state. **This should always be done.** It is important to be able to identify later what the cause of read/write failures was, which some other interfaces in the library check.

It is important to reiterate that in both Read and Write methods a return value of 0 is not a failure/error. Both methods return the number of bytes read for the message body while some messages have no message body. Only a return value of -1 indicates error, so make sure to return this value in the event of failure.

5.2.1 Example - Using a Message

```
Stream packet;
SetComponentAuthority message, messageCopy;

// Populate a message with data
message.SetDestinationID( Address(1, 1, 1, 1) );
message.SetSourceID( Address( 1, 2, 1, 1) );
message.SetAuthorityCode( 5 );

// Try to serialize the message
if( message.Write( packet ) )
{
    // Read the serialized message
    if( messageCopy.Read( packet ) )
    {
        if( message.GetDestinationID() == messageCopy.GetDestinationID() &&
            message.GetSourceID() == messageCopy.GetSourceID() &&
            message.GetAuthorityCode() == messageCopy.GetAuthorityCode() )
        {
            // Success!
            return JAUS_OK;
        }
        else
        {
            // Message data read from packet does not
            // match that of the original message structure.  There
            // must be an error in either Read, Write or both!
            return JAUS_FAILURE;
        }
    }
    else
    {
        // Failed to read message!
        return JAUS_FAILURE;
    }
}
else
{
    // Failed to write message
    return JAUS_FAILURE;
}
```

```
return JAUS_FAILURE;
```

2.5 LargeDataSet

Whenever a JAUS message contained in a Stream requires more than JAUS_MAX_PACKET_SIZE bytes to fit, it will need to be broken up into a multi-packet stream sequence following the rules for large data sets in JAUS. However, developers do not need to perform this step unless they disable automatic stream handling in the Component interface [3.1]. Before any Message based class can read JAUS data, a multi-packet stream must be merged into a single Stream. All of this can be accomplished using the LargeDataSet class. It has functions that will take a single "large" Stream, split the data into a multi-packet stream sequence, and then merge these individual packets back into a single Stream again even if the data is out of order.

2.6 Adding New Messages to Library

In the previous sections there have been examples of how to create a JAUS message using the Message class. This section lists the steps needed by developers to add a new message to the library.

For each new message you must follow a few steps. The first step is different if you are using Visual Studio 2005 versus Linux and gcc. In VS2005, you must add your files to the jmsglib project. For Linux you must edit the build/linux/jaus++/jmsglib Makefile or jmsglib project in CodeLite. Then depending on your message type, you can perform the remaining three steps which are:

a) For command class messages 0x0000-0x1FFF
   i) Add code to include/jaus/messages/command/commandcodes.h
   ii) Add include line to include/jaus/messages/command/commandmessages.h
   iii) Add the creation of your message switch statement in the CreateCommandMessage function in src/jaus/messages/messagecreator.cpp
b) For query class messages 0x2000-0x3FFF
   i) Add code to include/jaus/messages/query/querycodes.h
   ii) Add include line to include/jaus/messages/query/querymessages.h
   iii) Add the creation of your message switch statement in the CreateQueryMessage function in src/jaus/messages/messagecreator.cpp
   iv) Add the matching response code generated by your query message to the GetQueryResponseType function src/jaus/messages/messagecreator.cpp
c) For inform class messages 0x4000-0x5FFF
   i) Add code to include/jaus/messages/inform/informcodes.h
   ii) Add include line to include/jaus/messages/inform/informmessages.h
   iii) Add the creation of your message switch statement in the CreateInformMessage function in src/jaus/messages/messagecreator.cpp
   iv) If your inform message is in direct response to a query message, then you must add the capability to automatically identify the query associated with your report by adding to the GetInformQueryType function in src/jaus/messages/messagecreator.cpp
d) For experimental messages 0xD000-0xFFFF

i) Add code to include/jaus/messages/user/experimentalcodes.h
ii) Add include line to include/jaus/messages/user/experimentalmessages.h
iii) Add the creation of your message switch statement in the CreateExperimentalMessage function in src/jaus/messages/messagecreator.cpp

In the case of a Command or Query message you must also add the response codes to your message to the GetResponseCodes method and for Inform messages to GetInformQueryType method in src/jaus/messages/messagecreator.cpp. This is needed for automatic lookup of response types to messages sent.

Once you have followed these two steps, rebuild the JAUS++ libraries, and it will be added. If done correctly, you can use the static functions of MessageCreator to create an instance of your new message by only providing a message code. This may not sound all that useful at first, but if you are parsing messages and all you have is the message type information, you can create an instance of the message you need and read the data with a few lines of code. This type of operation is used by some messages like SpoolMission, CreateEvent, and the Node Manager.

In the current version of this library, the above steps are the only way to add new messages. However, if you don't plan to use MessageCreator with your new message, this does not matter.

2.7 Conclusion

This section covered the primary classes defined within the JAUS++ Message Library, which are the building blocks for all communication. To summarize, JAUS messages are created by inheriting from the Message class. They store all the raw data associated with a message. The raw data includes converted Scaled Integers and other data read from serialized JAUS messages (byte array data conforming to JAUS standard). All values stored in Message classes are serialized and de-serialized using a Stream object. A Stream contains an entire serialized JAUS message (message header and data block). A single Stream that is larger than JAUS_MAX_PACKET_SIZE (4095 bytes as of this writing) can be manually split into a multi-packet stream sequence using the LargeDataSet class. Stream data that is part of a multi-packet stream sequence is also collected and merged back into a single data block using LargeDataSet. Figure 2 of Appendix A illustrates all of these concepts. However, use of the LargeDataSet class is optional because all multi-packet sequences are processed automatically within the Components Library.

Node Manager

The Node Manager is the most critical part of any JAUS application you are developing. It is the "central hub" for all communication on a node, handling message routing between components on the same machine and across a network. Without a Node Manager running, no communication can take place at all. Therefore, before we describe how to create a component and send/receive messages, you must know how to create/run a Node Manager. None of the example component program will run until a Node Manager is detected on your node.

2.6 Configuration Options

The Node Manager can be run as a standalone application (e.g. wxNodeManager), or within your application. The NodeManager class contains all the functionality needed for initialization and setup of a complete Node Manager. Communication to other components using the Node Manager is done using the Component class 3.1 using shared memory. The Node Manager handles communication to other nodes using UDP, TCP, or Serial transports. Therefore, as long as you have a Node Manager running on each Node, components can communicate with each other (Figure 3 - Component Communication). For a complete example of how to create a Node Manager, see the example_nodemanager.cpp program.

Since it is unlikely that developers will want to "hard-code" the configuration of the node manager in applications, the NodeManager class can be initialized from an XML file. The following describes the format and all the options that can be used in an XML configuration file:

```
<?xml version="1.0" standalone="yes" ?>
<Jaus>
    <!-- The broadcast attribute indicates that the Node should broadcast
         hearbeat messages to 255.255.1.1 when the value is true.  Otherwise
         it only broadcasts to SubsystemID.255.1.1 which is used for Node
         discovery.  See Dynamic Discovery in the JAUS RA for more info about
         discovery procedures.-->
    <NodeManagerComponent broadcast="true">
        <!—Header to add to transport layer (UDP/TCP)-->
        <NetHeader>JAUS01.0</NetHeader>
        <!—Port Number for TCP/UDP Communication-->
        <NetPort>3794</NetPort>
        <SubsystemID>1</SubsystemID>
        <NodeID>2</NodeID>
        <!-- Option Subsystem Identification data if available.  Authority
             is the lowest level of authority to gain control of the
             subsystem, identification is the name of the subsystem and
             type is the type of subsystem, however the values are
             not defined as of this writing (JAUS RA 3.3)-->
        <SubsystemIdentification authority="0"
                                 identification="Subsystem Type String"
                                 type="10001"/>
             <!-- Enable or disable logging at startup by setting the
                  logdata attribute to true, use false to not have logging
                  at startup. Log files contain information such as type of
                  type of message received by node, what transport layer it
                  was received on, and message header data. Log file names
                  are unique and based on time stamp information. -->
        <NodeLogger logdata="false" />
```

```
        <!-- This value represents how large a block of shared memory you
             want to use for storing messages coming in to the Node Manager
             for routing.  The value is in bytes. Default size is
             2 MB.  If you plan to stream video from your node, you should
             make this value larger based on image resolution. -->
        <MessageBoxSize>2097152</MessageBoxSize>
        <!-- Transport indicates that the default
             transport medium for inter-node communication
             should be.  Supported values are UDP or TCP-->
        <Transport>UDP</Transport>
        <!-- Set the multicast address to use for multicast transmissions. If
             field is not present, a value of 224.1.0.1 is used by
             default.-->
        <MulticastAddress>224.1.0.1</ MulticastAddress >
        <NodeConnections>
            <!-- If a node can not be found using dynamic discovery, then
                 you can add connections to nodes
                 -->
            <Connection subsystem="1" node="2" host="128.185.3.1" />
            <Connection subsystem="1" node="3" port="com2" baud="9600"
                        bits="8" parity="0" stop="1"/>
        </NodeConnections>
    </NodeManagerComponent>
</Jaus>
```

See the bin/settings/nodesettings.xml file for an example configuration file for a Node Manager.

2.7 wxNodeManager

Included with JAUS++ is a wxWidgets-based GUI version of the Node Manager. wxWidgets is an open source API that enables users to develop GUIs for multiple operating systems. This permits wxNodeManager to be used in various environments. For developers looking to modify the interfaces please reference the wxWidgets documentation to be able to work within the wxNodeManager code. The latest release of wxWidgets (as of this writing) can be downloaded at http://www.wxwidgets.org/downloads/. The wxNodeManager has been verified to work with version 2.8.8 of the library.

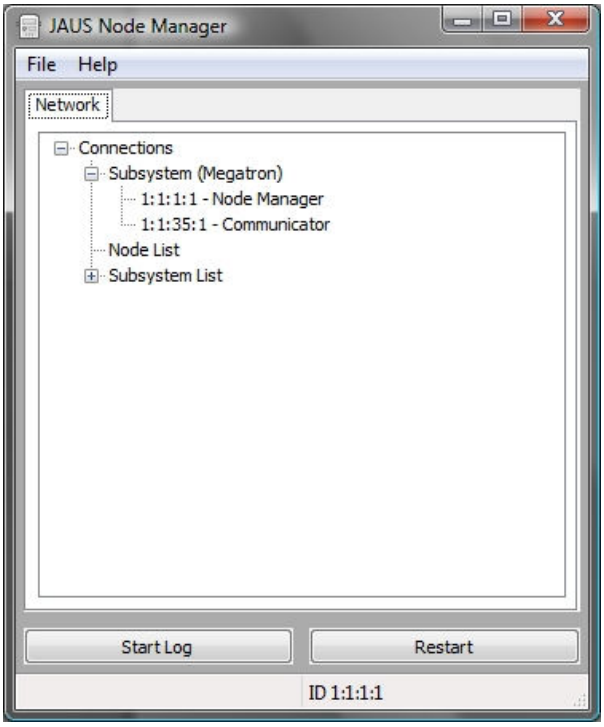


Figure 1 - wxNodeManager

The wxNodeManager provides a simple interface that lists the known subsystem configuration (for the subsystem the node belongs to), a list of all discovered nodes, and a subsystem list of all components broadcasting to 255.255.1.1 for dynamic discovery. From the File menu it is possible to load an XML configuration file, change the ID of the node, and toggle subsystem broadcasting to 255.255.1.1 from the node. The "Start Log" button will create a log file in the logs directory and will keep logging message traffic until pressed again. The "Restart" button will restart the node manager with the same configuration data previously used.

3 Components

This section provides an overview of using the JAUS++ library to communicate with JAUS components. In JAUS, a component represents the lowest level of operations within the reference architecture. Components perform a single cohesive function that has been broken down to a primitive function that would not be advantageous to simplify further.

3.1 Component

The Component class is the lowest level class interface provided to developers for creating a component. It provides an interface to the Node Manager, (NodeManager), for sending and receiving messages, and implements the core set of JAUS messages. By providing a complete implementation of the core set of JAUS messages within the Component interface, ANY class that inherits from Component is fully JAUS compliant. However, for those who wish for more control of things, it is still possible to override this behavior through the use of virtual functions. The full class definition of Component can be found in the include/jaus/components/component.h file. The examples describing how to use the Component class are contained in the src/examples/example_component.cpp example program.

3.2 Inter-Process Communication

The primary method of inter-process communication used in the JAUS++ library is through shared memory. Similar interfaces for creating and using shared memory exist in both Windows and Linux operating systems, so it became the primary choice for communication between components and the Node Manager. Each component has two shared memory buffers, created using JSharedMemory, one as in an inbox for incoming messages, and the other a view of the inbox for the Node Manager (component outbox). All serialized JAUS messages are transferred to and from a component using these mapped memory locations. Developers do not have direct access to these memory locations and must use the methods built into the Component interface. If communication is taking place between two components on the same node, the Node Manager transfers the message data directly. This means that serialized JAUS messages greater than the 4095 bytes do not need to be broken down into multi-packet streams following the rules of Large Data Sets in the JAUS RA. Large messages are only converted to multi-packet sequences when transmitting to a component on another node. This makes data transfer more efficient, but depending on the size and amount of traffic being used requires more memory.

3.3 Component Identification

It is important to note that in Part 1 of the JAUS RA several component IDs are defined. For example, all Velocity State Sensors will have a component ID of 42 (i.e. 1:1:42:1). It is important that if users create custom components they ensure they don't use an already reserved component ID.

3.4 Component Initialization

Initializing a component makes it capable of receiving messages and connecting to the Node Manager. It must be done before you will be able to do anything, so don't forget to do it. Second, you must have a Node Manager running before any communication can take place with other components. You can instantiate a Node Manager within your application or run it as a

separate program (like wxNodeManager). As long a Node Manager program is running on your machine, and your component is initialized with the same Subsystem and Node ID, you'll be able to communicate with other components. The Initialize function requires three parameters: an ASCII string representing the component name, an Address structure for the ID of the component, and how large in bytes the components shared memory should be.

The parameter describing the size of the components shared memory has a default size which is large enough to hold 100 full size JAUS messages (4096 bytes each). It is up to the developer to make this size smaller or larger depending on the type/amount of message traffic they expect to have. For example, if your component is transmitting images, with each image being 2MB in size, then you'll want to create a larger shared memory buffer to hold it.

The JAUS ID parameter is necessary and currently cannot be chosen dynamically. It is up to the developer to determine subsystem and node ID values on their own. However, it is possible to check if a node manager is running on the host machine and acquire its' ID using the static function Component::IsNodeManagerPresent(Address* id). The following example demonstrates how to discover the Node Manager on the host system and get its ID value.

This example shows the Component using a while loop to check if the Node Manager is present. If it is not present the component sleeps for 100ms before trying again. This continues until the user sets the exit flag or a Node Manager is found.

```
Address nodeID;     // ID of the node manager.

cout << "Looking for node manager...";

//  Keep looping until a Node Manager is
//  initialized and we can connect to it.
while(exitFlag == false)
{
    //  The IsNodeMangerPresent parameter is a pointer to an Address
    //  structure. Use NULL if you don't care about the Node Manager
    //  ID, or pass an argument to get a copy of the ID.
    if(Component::IsNodeManagerPresent(&nodeID))
    {
        cout << "Success!\n";
        cout << "Node Manager ID is: ";
        nodeID.PrintID();
        break;
    }
    //  Keep looping. (uses Sleep to avoid maxing out CPU)
    Sleep(100);
}
//  Report failure on invalid Node Manager ID.
if(nodeID.IsValid() == false)
{
    cout << "Failure.\n";
    cout << "Exiting...";
    return 0;
}
```

It is possible to initialize a component when no Node Manager is running. Once the Node Manager starts, the component will be detected automatically if the subsystem and node ID values of your component matches the Node Manager subsystem and node ID. You can check to see if your component is connected to a Node Manager with the IsConnected() function. If another component is already present on your system with the same JAUS ID, then initialization will fail. The following example demonstrates how to initialize an instance of a Component and declare the status is ready.

```
Component component;

cout << "Initializing component...";
if(component.Initialize("My Example Component",
                        Address(nodeID.mSubsystem, nodeID.mNode, 2, 1)))
{
     cout << "Success!\n";
     // Wait until connected to node manager.
     // The node manger included with JAUS++ will automatically
     // identify running components and connect to them.
     while(!component.IsConnected())
     {
         Sleep(100);
     }

     // If the component is ready, set the status to ready.
     component.SetPrimaryStatus(Component::Status::Ready);

     // Component is now ready and operations can be used after
     // checking status of node manager connection. The remaining
     // sections in this chapter's code would be placed here in
     // a real component.

 }
 else
 {
     cout << "Failure!\n";
     // Display the reason for failure.
     component.PrintJausError();
 }

 // Shutdown the component.
 component.Shutdown();
```

3.5 Component Communications

To reiterate, a Component in JAUS must perform a single cohesive function which has already been broken down to a point where it is not worthwhile to break down further. With that in mind, once connected to a Node Manager the Component can begin to send and receive messages to perform its' specific function.

At this point the Component as initialized in the previous section is ready, and messages can be sent after checking the status of the connection with the Node Manager.

5.1.1 Example - Component Sending a Message (Non-Blocking)

A simple example to show how to send a message can be done with the Query Services message. This message allows a Component to request the capabilities of another Component. This example shows how to set the required source and destination values for the message to be sent. QueryServices, as well as all other messages, uses the methods of the its' parent class Message to set Ack/Nack requests (in this example an acknowledge request is made), set the Destination ID of the message (in this example the Node Manager), and set the Source ID (in this case by retrieving the ID from the Component Class).

Any responses that are generated by the receiving components will be sent to any registered callback messages or the ProcessCommandMessage, ProcessQueryMessage, ProcessInformMessage, ProcessAckNackMessage, or ProcessExperimentalMessage virtual functions of the Component class depending on the type of message received.

If the messages are sent successfully, responses from the receiving component (in this example, the Node Manager) will be sent to the component and passed to the ProcessAckNackMessage, then ProcessInformMessage method of the Component class. This is because the query sent requested Acknowledgement, and the response is an Inform message. Users should inherit from the Component class and overload these functions to add additional functionality.

```
QueryHeartbeatPulse queryHeartbeatPulse;

queryHeartbeatPulse.SetDestinationID(nodeID);
queryHeartbeatPulse.SetSourceID(component.GetID());
queryHeartbeatPulse.SetAckNack(Header::AckNack::Request);

cout << "Sending Query without blocking...";
if(component.Send(&queryHeartbeatPulse) == JAUS_OK)
{
    cout << "Success!\n";
}
else
{
    cout << "Failure.\n";
    break;
}
```

5.2.1 Example - Component Sending/Receiving a Message (Blocking)

In this example the same message is sent as in the previous example, but uses an overloaded member of the Send function to provide a receipt upon sending. Blocking causes the program to wait for a response message before continuing, thus allowing you wait until the response is received. The response message in this example is stored in a receipt structure.

```
while(gExitFlag == false && component.IsConnected())
{
    QueryServices queryServices;
    QueryHeartbeatPulse queryHeartbeatPulse;
    Receipt receipt;
```

```
        queryServices.SetAckNack(Header::AckNack::Request);
        queryServices.SetDestinationID(nodeID);
        queryServices.SetSourceID(component.GetID());

        // Note: When the receipt is deleted (or goes out of scope)
        // the message data stored within it will be deleted also.
        cout << "Sending Query message with receipt (blocking)...";
        if(component.Send(&queryServices, receipt) == JAUS_OK)
        {
            // At this point the response message is stored in
            // the Receipt (receipt) which has other information such
            // as how long it took to get the response, etc.

            if(receipt.ReceivedAcknowledge() &&
               receipt.GetSendCount() == 1)
            {
                cout << "Success!\n";

                // Display received data to console.
                const ReportServices* report = dynamic_cast<const
                      ReportServices*>(receipt.GetResponseMessage());
                report->Print();
            }
            else
            {
                cout << "Failure.\n";
                break;
            }
        else
        {
            cout << "Failure.\n";
        }
    }
```

5.3.1 Message Callbacks

It is possible to use a callback to process specific types of messages received by a component. This can be done through inheritance of the MessageCallback class, or using a Function Callback. Callbacks are added to the Component using the RegisterCallback methods, and removed using the RemoveCallback methods. An example using a Function Callback is shown in src/examples/example_component.cpp source file.

5.4.1 Example – Receiving Messages through Inheritance

The final option that can be used to handle a message received by a component is through inheritance. By inheriting from the Component class, you can overload any of the virtual Process methods (e.g. ProcessQueryMessage, ProcessCommandMessage). This is the recommended method for creating more complex components. Remember, by default the Component class supports handling of all Core JAUS messages (e.g. Set Component Authority, Query Authority, Request Component Control), so if you overload one of the Process methods,

and it contains a message you are not interested in, call the parent classes implementation as shown in the following example:

```
int PrimitiveDriver::ProcessQueryMessage(const Message* msg)
{
    int result = JAUS_FAILURE;
    // Based on the type of query message handle it
    // ourself, or pass to the parent class
    switch(msg->GetCommandCode())
    {
    case JAUS_QUERY_WRENCH_EFFORT:
        {
            // Cast to a QueryWrenchEffort message, and the
            // Generate a response
            const QueryWrenchEffort* query = dynamic_cast<const
                                    QueryWrenchEffort*>(msg);
            if(query)
            {
                // Send response message with current wrench effort

                result = JAUS_OK;
            }
        }
        break;
    default:
        // By default, have the parent class process the query message
        // because it may be able to handle the message.
        result = Component::ProcessQueryMessage(msg);
        break;
    }

    return result;
}
```

In the above example, a class called PrimitiveDriver has inherited from the Component class and overloaded the ProcessQueryMessage method. If it is capable of processing the message, it does so, otherwise it lets the parent class (in this case Component) process the message. By doing this it is possible to build more advance component classes that can support many messages. Examples of this can be seen in src/examples/example_primitivedriver.cpp. Examples of more advanced component interfaces are the SubscriberComponent, InformComponent, and CommandComponent classes.

3.6 Creating a Component

Creating a new JAUS component is simple to do by building from interfaces within the Component Library. There are four component classes to inherit from, Component, SubscriberComponent, InformComponent, and CommandComponent. As mentioned in the previous sections, Component is the lowest level class you can inherit from with the others building inheriting from it in the order listed previously.

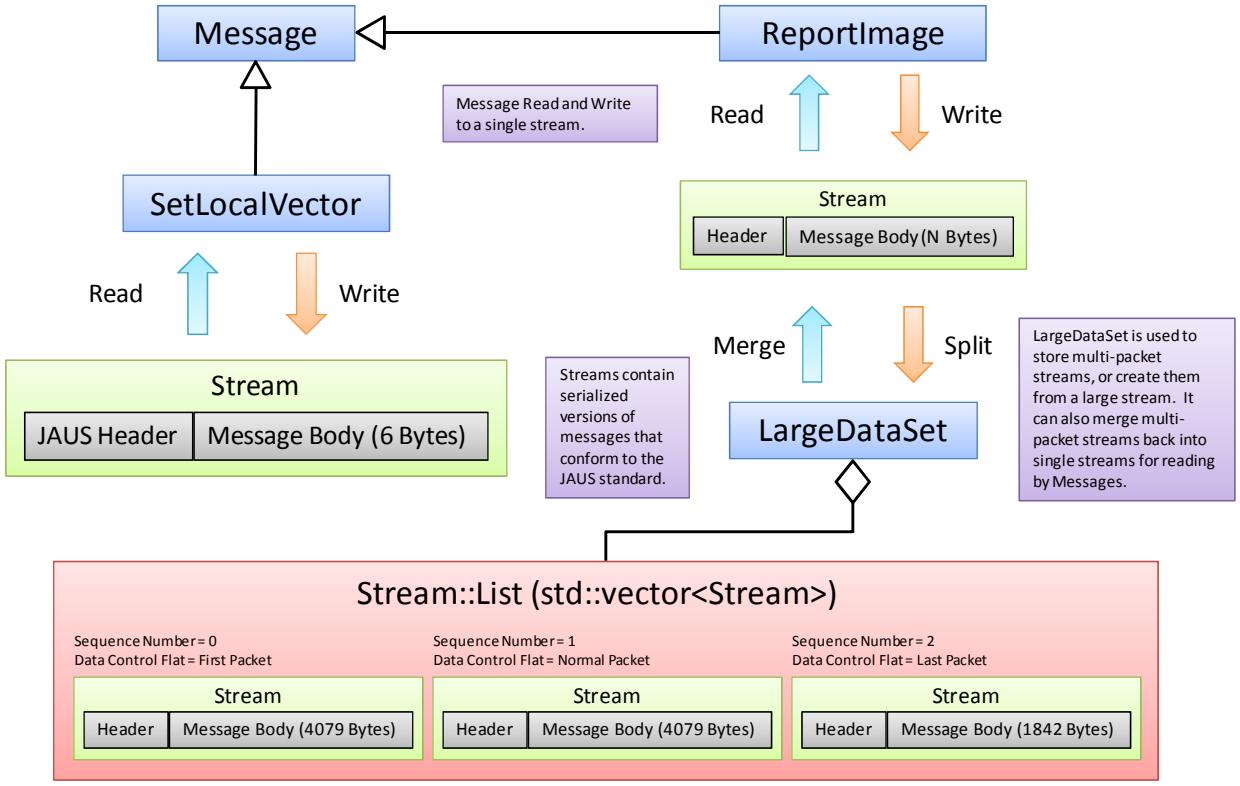
For example, if you are creating a component that just needs to gather information from other components, you would inherit from SubscriberComponent because it has methods for

maintenance of Service Connections and Event subscriptions you will make. If you are creating a sensor which will be providing information to other components, inherit from the InformComponent class because it has methods for maintaining Service Connections and Events generated by your component. Finally, if you need to perform any of the previous actions, plus command other components, you would inherit from the CommandComponent class because it has the same features of the previous interfaces plus it can keep track of components you have control over.

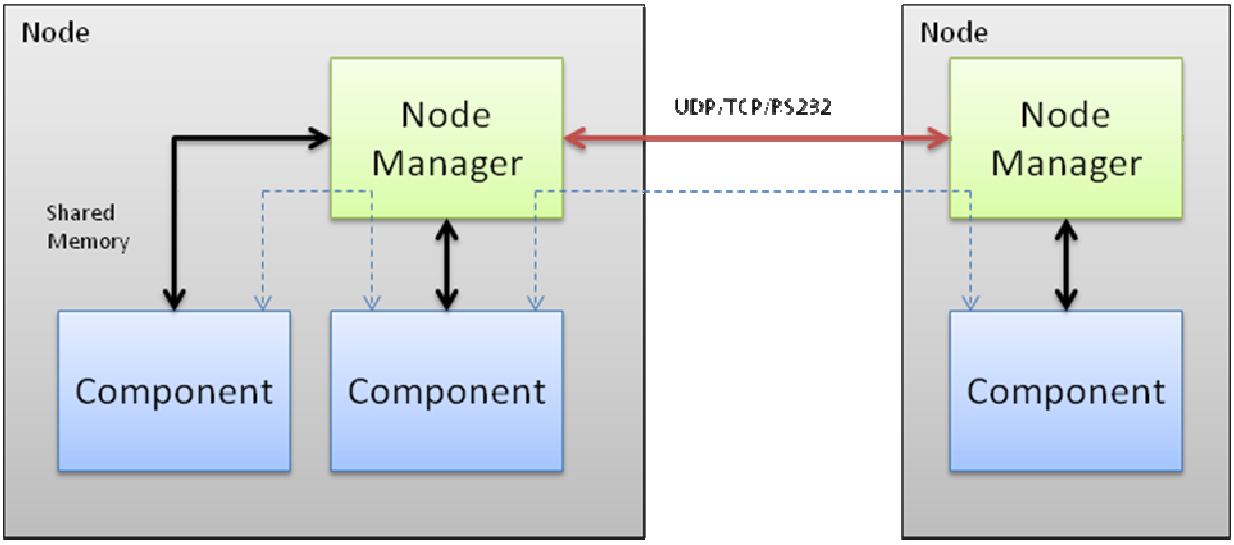
No matter which component class you derive from, you must make sure you overload the SetupService method. This method is called when a component is initialized and sets what input and output messages (service) your component will support. The JAUS++ library comes with many examples which explain how to create components. Specifically, the Services Library implements many of the Components/Services defined by the JAUS Service Set (e.g. Global Pose Sensor, Velocity State Sensor, Primitive Driver, and Global Vector Driver). All of these examples are fully commented and documented and are the best way to learn how to create a new component.

4 Appendix

4.1 Figure 2 - Message Structures



4.2 Figure 3 - Component Communication



4.3 Table 1 - Shared Memory Message Queue Format

Field #	Name	Type	Units	Interpretation
1	Length	UINT32	Bytes	Size of shared memory in bytes, including 4 bytes for this field.
2	Write Time	UINT32	Milliseconds	Time of last writing of message to shared memory in milliseconds. All time is UTC. This value is equal to milliseconds + sec*1000 + min*60000 + hour*3600000.
3	Read Time	UINT32	Milliseconds	Time of last read/removal of message from shared memory in milliseconds. All time is UTC. This value is equal to milliseconds + sec*1000 + min*60000 + hour*3600000.
4	Count	UINT32	Units	Number of messages streams stored in shared memory box.
5	Start Position	UINT32	Byte	Byte position of first JAUS message in shared memory. If no messages in shared memory, this value will be 24.
6	End Position	UINT32	Byte	Byte position of the end of the last JAUS message in shared memory. If no messages in shared memory this value is 24.
7	Message Size	UINT32	Bytes	Size of the following message in bytes. The smallest value is 16 which is the size of a JAUS Header.
8	JAUS Message Stream	N/A	N/A	A serialized JAUS message. Format is identical to messages stored in Message Streams with 1 JAUS Header, followed by message body data. Size is determined by previous field (Message Size)
…				
N	Message Size	UINT32	Bytes	Size of the following message in bytes. The smallest value is 16 which is the size of a JAUS Header.

Field #	Name	Type	Units	Interpretation
N+1	JAUS Message Stream	N/A	N/A	A serialized JAUS message. Format is identical to messages stored in Message Streams with 1 JAUS Header, followed by message body data. Size is determined by previous field (Message Size)