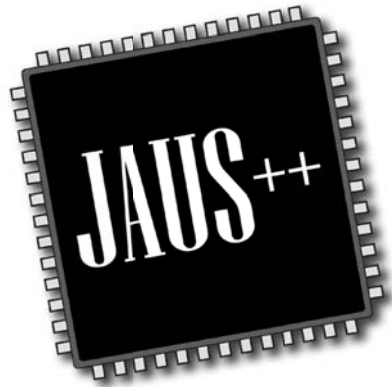


VERSION 2.0

USER DOCUMENTATION



1	Joint Architecture for Unmanned Systems (JAUS).....	4
1.1	RECOMMENDED Reading	4
1.2	License	4
1.3	Common Vocabulary	4
1.4	JAUS++ Directory File Structure.....	5
1.5	JAUS++ Libraries and Applications.....	6
1.5.1	CORE LIBRARY (LIBJAUSCORE).....	6
1.5.2	Mobility Library (LIBJAUSMOBILITY).....	6
1.5.3	Extras Library (LIBJAUSEXTRAS).....	6
1.5.4	Video Client	6
2	Installation	7
2.1	Windows Installer	7
2.2	CMake.....	7
2.3	Namespaces.....	11
2.4	Class Names	11
2.5	Class Methods.....	11
2.6	Class Members.....	11
2.7	Functions/Methods.....	11
2.8	Enumerations.....	12
2.9	Constants	12
3	CORE Library	13
3.1	Primitive Types.....	13
3.2	Address	13
3.3	Header	13
3.4	Packet	13
3.5	Message.....	14
3.5.1	Example - How to Create a Message Class.....	14

- 3.6 Services17
 - 3.6.1 Creating a Service17
- 3.7 Components.....17
- 3.8 Conclusion.....17

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 service-based message passing architecture that describes data fields and services. This standard is designed to be 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>.

This version of JAUS++ is based off of the SAE Standards AS5669A, AS5710, and AS6009. If you are interested in the JAUS Reference Architecture (RA) 3.X version of JAUS++, please see the 1.5 version hosted of this library at <http://active-ist.sourceforge.net>.

1.1 RECOMMENDED READING

Users are advised to look over the SAE JAUS standards documents first. They provide a more detailed overview (i.e. vocabulary, concepts, etc.). As of this writing the current documents used in this software are the AS5669A, AS5710, and AS6009 standards. An additional reference is the Unmanned Systems and Robotics Interoperability Center’s website (<http://www.usaric.org/>).

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.

1.2 LICENSE

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

1.3 COMMON VOCABULARY

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

- JAUS RA – JAUS Reference Architecture or standards.
- SAE-JAUS – Society of Automotive Engineers (SAE) JAUS Standard
- Service – A service defines a complete set of functionality. This functionality is the ability to respond to or generated a specific set of messages (e.g. command/control, events).
- Component – A component is the lowest level of the system topology in JAUS. In this document a component is part of a software application that provides one or more services. Services provided can be GPS information, platform configuration data, or commands to other components, etc.

- 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.) and essentially maps to a computer on your robot.
- 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))
- System – A system is a collection of subsystems and is the highest level in the JAUS topology.
- 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 JAUS. 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 if you plan to develop new messages.

1.4 JAUS++ DIRECTORY FILE STRUCTURE

The following describes the folder/directory structure for JAUS++ code.

- jaus++
 - <version #>
 - include – Contains all header files for library
 - jaus – Main folder containing all files related to the libraries comprising JAUS++
 - core – Folder containing files for the core service set library and common interfaces for creating components
 - control – Access Control service files and messages
 - discovery – Discovery service files and messages
 - events – Events service files and messages
 - liveness – Liveness service files and messages
 - management – Management service files and messages
 - time – Time service files and messages
 - transport – Transport service files and messages
 - mobility – Folder containing files for the mobility service set library
 - drivers – Contains driving services and messages (e.g. Primitive Driver, Global Waypoint Driver)
 - list – List Manager service and messages
 - sensors – Contains mobility sensor services and messages (e.g. Global Pose Sensor, Velocity State Sensor)
 - extras – Folder containing custom or experimental services that do not conform to the SAE-JAUS standard
 - programs – Contains header and include files for example and other tool programs
 - 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
 - msvc9 – Microsoft Visual Studio 8 (2005) solutions
 - jaus++ – Builds main JAUS++ library and programs
 - linux – Contains Linux Makefiles for compilation if available
 - codelite – Contains CodeLite workspaces and project files
 - ext – Contains any external libraries needed to build JAUS++ library

1.5 JAUS++ LIBRARIES AND APPLICATIONS

1.5.1 CORE LIBRARY (LIBJAUSCORE)

The library is an implementation of the JAUS Transport Layer and Core Service Set standards. It contains the base classes for deriving messages, services, and components that all other libraries build upon.

1.5.2 MOBILITY LIBRARY (LIBJAUSMOBILITY)

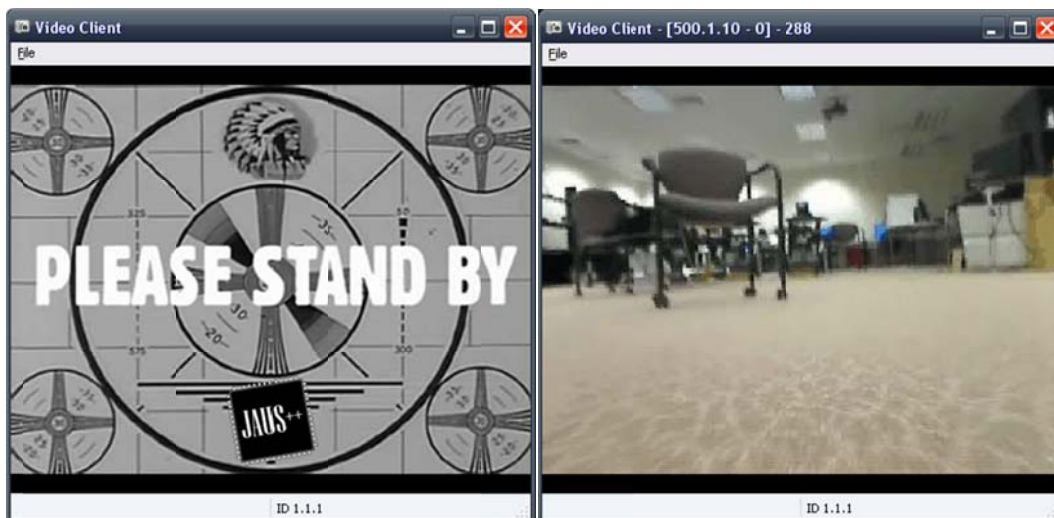
This library is an implementation of the JAUS Mobility Service Set. It contains data structures for all mobility messages, and select services (e.g. Global Pose Sensor, Global Waypoint Driver).

1.5.3 EXTRAS LIBRARY (LIBJAUSEXTRAS)

This library contains custom and experimental services and messages that are not part of the official JAUS standard. As new JAUS services are published by SAE, any duplicate functionality within this library will be deprecated and removed. This library exists to help developers fill the gap of missing services that they need.

1.5.4 VIDEO CLIENT

Video Client is a simple GUI for subscribing to video feeds of platforms using the Visual Sensor service of LIBJAUSEXTRAS. It also has the ability to use a Joystick to teleoperate a Primitive Driver service of that vehicle as well. Build this program requires the wxWidgets library.



2 INSTALLATION

All platforms that use this library must have the CxUtils library installed. JAUS++ is distributed with CxUtils, so you should have it with your download. **When building JAUS++, it will build CxUtils as well, so you only need to follow the external dependency and post-build instructions of the CxUtils Documentation (provided with CxUtils) if you encounter any problems.**

2.1 WINDOWS INSTALLER

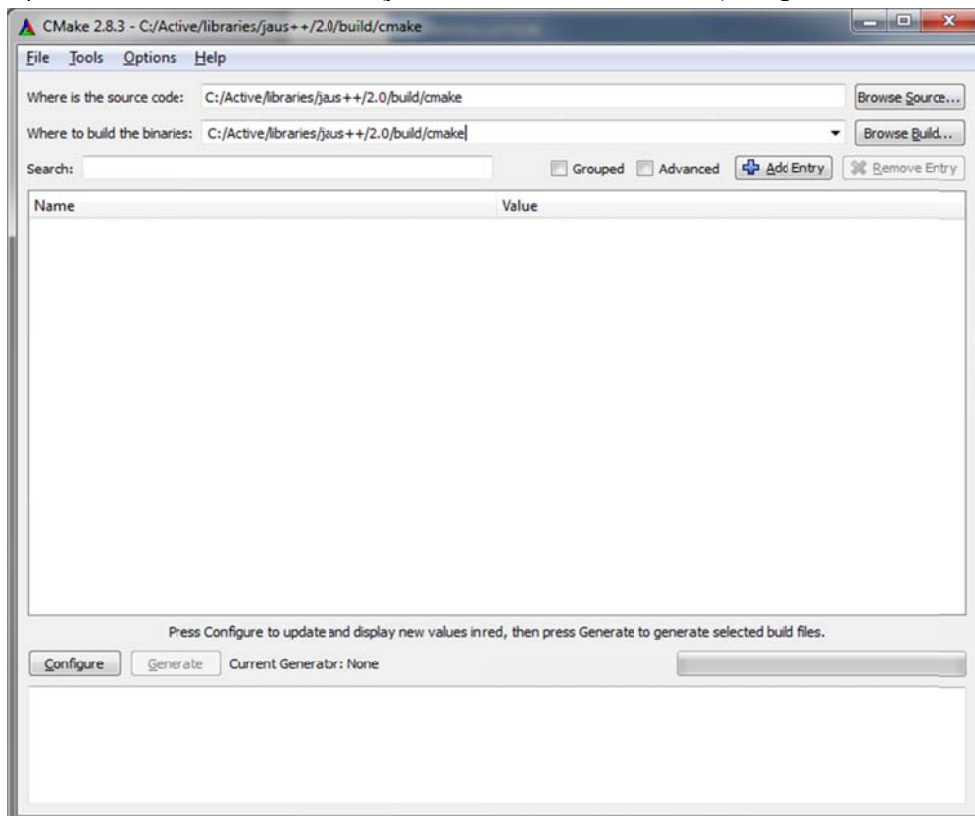
There is currently an installer for windows that will provide precompiled binaries of JAUS++ and the source code as well. If you plan to re-build the binaries yourself or make code changes, then make sure you install the library to a folder you have write permissions to (not Program Files) to avoid any problems.

2.2 CMAKE

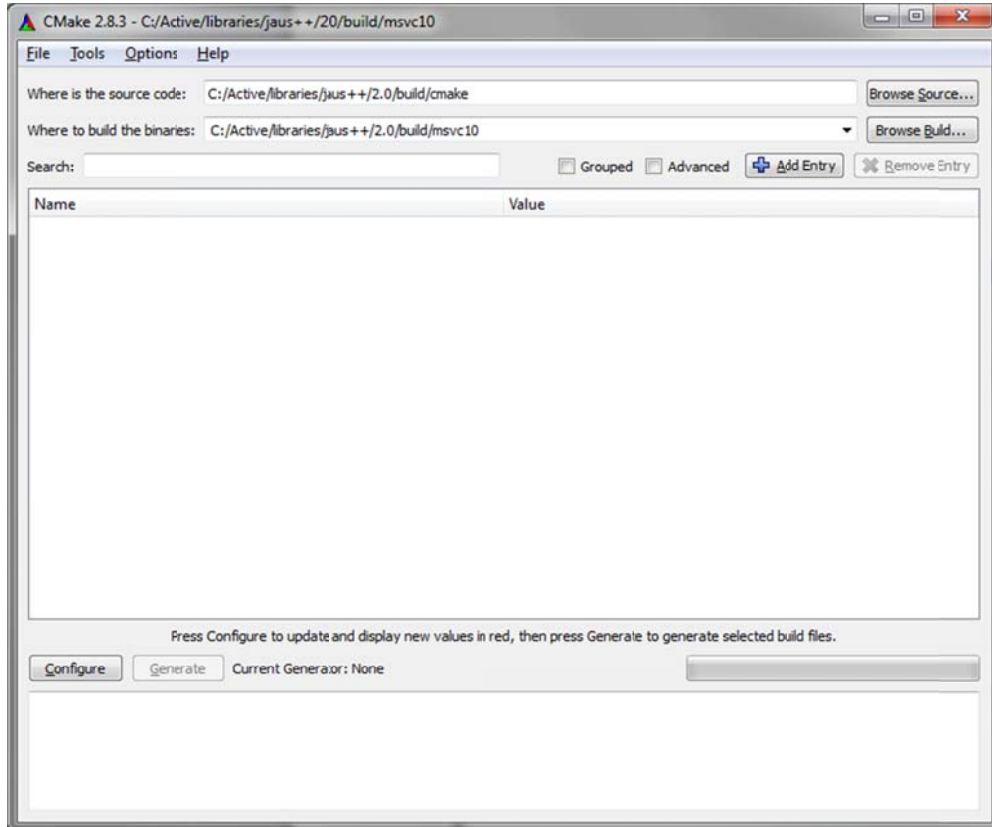
There are Visual Studio and CodeLite workspace/project files provided with JAUS++ that can be used to build the libraries and perform installation. However, there are CMake files provided so that you can create traditional Makefiles and install on *NIX systems more easily, or to create new Visual Studio project files for a different version (like VS2010).

To build using Visual Studio with CMake

- 1) Open the main CMakeLists.txt file (jaus++/<version>/build/cmake) using CMake like below:



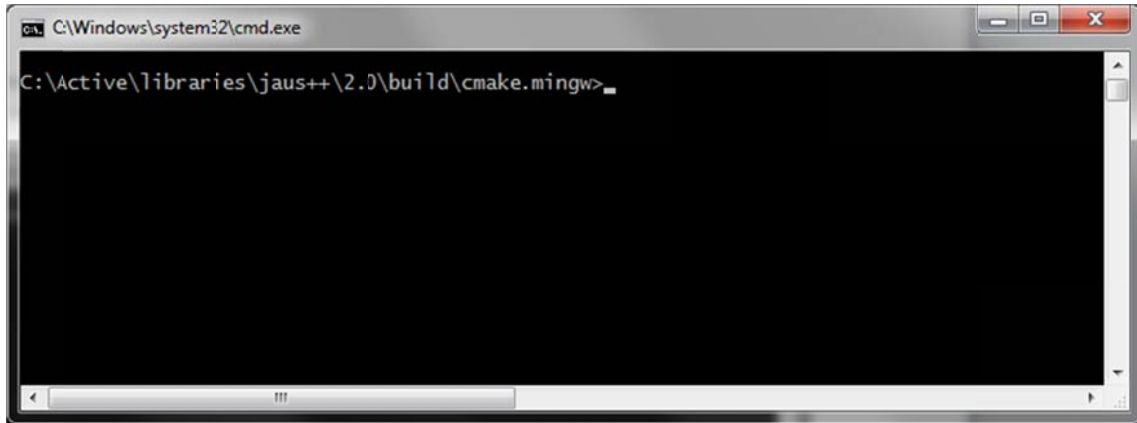
- 2) Change the “Where to build the binaries:” line to a different folder than cmake (e.g. like the version of Visual Studio (msvc10):



- 3) Hit the “Configure” button and select the version of Visual Studio you want to use
- 4) Check any additional options or change fields if needed. If wxWidgets is not found, that is OK, you will just not be able to build the Video Client GUI
- 5) Hit the “Configure” button again until you can press the “Generate” button
- 6) Hit the “Generate” button
- 7) Go to the location of the “Where to build the binaries” that you selected is, and open the Visual Studio solution file.
- 8) Build the SDK

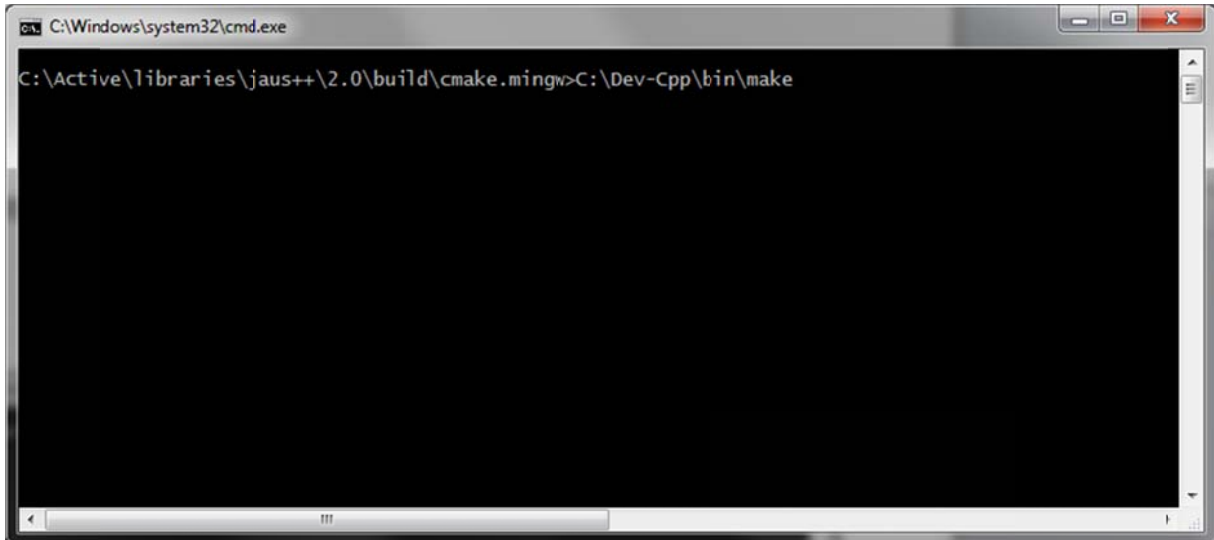
To build using MinGW with CMake

- 1) Open the main CMakeLists.txt file (jaus++/<version>/build/cmake) using CMake like above example for VS
- 2) Change the “Where to build the binaries:” line to a different output folder than (jaus++/<version>/build/cmake (e.g. like jaus++/<version>/build/mingw)
- 3) Hit the “Configure” button and select the version of MinGW Makefiles
- 4) Check any additional options or change fields if needed. If wxWidgets is not found, that is OK, you will just not be able to build the Video Client GUI
- 5) Hit the “Configure” button again until you can press the “Generate” button
- 6) Hit the “Generate” button
- 7) Go to the location of the “Where to build the binaries” that you selected is using a command line like in the following image.



```
C:\Windows\system32\cmd.exe
C:\Active\libraries\jaus+\2.0\build\cmake.mingw>
```

- 8) Build by typing make. If this fails, then you don't have the path set to MinGW correctly, so you may need to type the full path as follows:



```
C:\Windows\system32\cmd.exe
C:\Active\libraries\jaus+\2.0\build\cmake.mingw>C:\Dev-Cpp\bin\make
```

```
C:\Windows\system32\cmd.exe - C:\Dev-Cpp\bin\make
C:\Active\libraries\jaus++\2.0\build\cmake.mingw>C:\Dev-Cpp\bin\make
-- Added Package: CxUtils
-- Added Package: TinyXML
-- Added Package: JAUS
-- Could NOT find wxWidgets (missing: wxWidgets_FOUND)
-- Configuring done
-- Generating done
-- Build files have been written to: C:/Active/libraries/jaus++/2.0/build/cmake.mingw
[ 17%] Built target cxjpeg-6b
[ 22%] Built target zlib
[ 28%] Built target png
[ 41%] Built target cxutils
[ 42%] Built target tinyxml
Scanning dependencies of target jauscore
Linking CXX shared library C:\Active\libraries\jaus++\2.0\bin\libjauscore.dll
Info: resolving std::cout by linking to __imp__ZSt4cout (auto-import)
Info: resolving vtable for __cxxabiv1::__vmi_class_type_info by linking to __imp__ZTVN10__cxxabiv121
Info: resolving vtable for __cxxabiv1::__class_type_info by linking to __imp__ZTVN10__cxxabiv117__cla
Info: resolving vtable for __cxxabiv1::__si_class_type_info by linking to __imp__ZTVN10__cxxabiv120__
Info: resolving vtable for std::exception by linking to __imp__ZTVSt9exception (auto-import)
Creating library file: C:\Active\libraries\jaus++\2.0\lib\libjauscore.dll.a
c:/mingw/bin/./lib/gcc/mingw32/4.5.0/./././././mingw32/bin/ld.exe: warning: auto-importing has been
This should work unless it involves constant data structures referencing symbols from auto-imported DL
[ 59%] Built target jauscore
Scanning dependencies of target jausmobility
Linking CXX shared library C:\Active\libraries\jaus++\2.0\bin\libjausmobility.dll
```

Notes for Windows

In for both Visual Studio and MinGW builds, the output files will go to the jaus++/<version>/bin and jaus++/<version>/lib folders. You will need to add these and any include folders to your IDE or Windows path for building/linking/ and running if not using a provided installer. Remember, if you get a missing DLL file at runtime, it is because you need to add the bin folder to your system path, or copy the DLL files to your .exe files directory.

To build and install using *NIX systems like Ubuntu follow the following steps:

1) Make sure minimum dependencies are installed on the system

```
sudo apt-get install libpng-dev libX11-dev libxtst-dev
```

2) Navigate to the jaus++/<version>/build/cmake folder

3) Type the following:

```
cmake .
```

```
make
```

```
sudo make install
```

4) Edit ld.so.conf to be able to find the libraries at runtime

```
sudo gedit /etc/ld.so.conf
```

5) Add the install path (default is /usr/local/lib/active) to the file, save and exit

6) Type the following to update runtime paths:

```
sudo ldconfig
```

7. Finished!

2.3 NAMESPACES

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

2.4 CLASS NAMES

All classes and structures are written in PascalCase.

Examples:

```
Address  
Header  
Packet  
LargeDataSet  
SetGlobalWaypoint
```

2.5 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();
```

2.6 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; ///  
unsigned char *mpArrayData; ///  
//< Regular data member.  
//< Pointer data members.
```

2.7 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)
```

2.8 ENUMERATIONS

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

Examples:

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

2.9 CONSTANTS

All global constants are written in capital letters with words separated with an underscore, '_'.

Examples:

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

3 CORE LIBRARY

This section provides an overview of the JAUS++ Core Service Set library. It describes the base types and classes for representing messages, services, and component. All other libraries are built off of the Core Library, as it defines all the Core Services required for basic functionality (e.g. Transport, Events, Discovery, Access Control, Management).

3.1 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.

3.2 ADDRESS

The Address class is used to store JAUS Component 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. All messages contain both source and destination IDs, and all components are created with an ID as well.

3.3 HEADER

The Header class contains all JAUS message header information. This includes: Message Properties (JAUS version, ACK/NACK, etc.), Data Size, Data Control Flag, Sequence Number, Source ID, and Destination ID. It is used primarily by the Transport services and you will probably not need to mess with it unless you are creating your own Transport layer.

3.4 PACKET

Besides Address and Header, Packet is the most commonly used class within this library. All serialized JAUS messages are stored in a Packet structure when sending or receiving. This is the version of a message that conforms to the JAUS standards. 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, Packet 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 to Little-Endian automatically. When writing to a Packet on a Big-Endian system, the data is converted to Little-Endian also, automatically.

Message based classes (3.5), read and write data using a Packet. A Packet with a complete JAUS message includes header information, followed by the message body. **Do not read/write more than one message header and data block to a packet.** If the length of the byte array data in a Packet is greater than the maximum size allowed by JAUS transport in use (current limit is 4095 bytes), do not worry. Software within this library is used to produce a Large Data Set (multi-packet stream) from an oversized Packet automatically. So you only need to write your message contacts to the packet, and let the interfaces handle format for transport.

3.5 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 (message code, source and destination ID, etc.), functions to access and set these values, and eleven (11) 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 for direct use by your services.

The functions that must be implemented are IsCommand, WriteMessageBody, ReadMessageBody, ClearMessageBody, Clone, GetPresenceVectorSize, GetPresenceVectorMask, GetMessageName, GetMessageCodeOfResponse, GetPresenceVector, and IsLargeDataSet. The WriteMessageBody and ReadMessageBody functions take a Packet object as their argument. The WriteMessageBody function serializes the message body data members to a Packet 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 Packet) can be done using the Read method of Message. All serialization (writing to a Packet) can be done using the Write method of Message. When writing, a Packet is cleared, a header is written, followed by message body data via the WriteMessageBody method. When reading, a Packet is not modified and its header is read, followed by the message body (using ReadMessageBody), and the results are saved to internal data members.

3.5.1 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 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 SetAuthority. It inherits from Message, and implements all the required methods. 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_CORE_DLL SetAuthority : public Message
{
public:
    SetAuthority(const Address& dest, const Address& src);
    SetAuthority(const SetAuthority& message);
    ~SetAuthority();
    inline void SetAuthorityCode(const Byte code;
    inline Byte GetAuthorityCode() const { return mAuthorityCode; }
    virtual bool IsCommand() const { return true; }
    virtual int WriteMessageBody(Packet& packet) const;
    virtual int ReadMessageBody(const Packet& packet);
    virtual Message* Clone() const;
    virtual UInt GetPresenceVector() const { return 0; }
    virtual UInt GetPresenceVectorSize() const { return 0; }
    virtual UInt GetPresenceVectorMask() const { return 0; }
    virtual UShort GetMessageCodeOfResponse() const { return 0; }
    virtual std::string GetMessageName() const { return "Set Authority"; }
    virtual void ClearMessageBody();
    virtual bool IsLargeDataSet(const unsigned int maxPayloadSize = 1437);
    virtual int RunTestCase() const;
    SetAuthority& operator=(const SetAuthority& message);
protected:
    Byte mAuthorityCode; ///< Authority value of sending component [0,255].
}
```

The Message class does not have a default default constructor. It has three parameter that must be set which include the message code (type), and destination and source ID of the message.

```
SetAuthority::SetAuthority(const Address& dest, const Address& src) :
    Message(SET_AUTHORITY, dest, src)
{
    mAuthorityCode = 0;
}
```

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* SetAuthority::Clone() const
{
    // Allocate memory for message.
    SetAuthority* ptr = new SetAuthority();
    // 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* SetAuthority::Clone() const
{
    return new SetAuthority(*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 message payload to the packet.
///
///  Message contents are written to the packet following the JAUS standard.
///
///  \param[out] packet Packet to write payload to.
///
///  \return -1 on error, otherwise number of bytes written.
///
////////////////////////////////////
int SetAuthority::WriteMessageBody(Packet& packet) const
{
    int expected = BYTE_SIZE;
    int written = 0;

    written += packet.Write(mAuthorityCode);

    return expected == written ? written : -1;
}

////////////////////////////////////
///
///  \brief Reads message payload from the packet.
///
///  Message contents are read from the packet following the JAUS standard.
///
///  \param[in] packet Packet containing message payload data to read.
///
///  \return -1 on error, otherwise number of bytes written.
///
////////////////////////////////////
int SetAuthority::ReadMessageBody(const Packet& packet)
{
    int expected = BYTE_SIZE;
    int read = 0;

```



```
    read += packet.Read(mAuthorityCode);  
  
    return expected == read ? read : -1;  
}
```

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.

3.6 SERVICES

With the current version of JAUS, the architecture has moved towards a service based framework. In this library, all message creation and manipulation is done within a Service derived from the Service class. The Service class contains several virtual methods that must be overloaded when creating a new service. Within the Core Library, all the Core Services are provided (e.g. Discovery, Events, Liveness). Depending on whether or not a Service inherits from another (e.g. Discovery inherits from Events), a service implementation may provide a class interface for C++ class inheritance. **Important note: class inheritance and service inheritance are not equivalent, although similar.** So when you create your own services, look at the implementation of the parent service you will inherit from to see if any interface is provided.

3.6.1 CREATING A SERVICE

When creating your own service, there are three primary methods you must overload: CreateMessage, Receive, and CheckServiceStatus. Any messages your service supports must be generated by the CreateMessage method. Failure to add creation of a message there will most likely result in the Transport service not being able to read the packet data, and dropping the message received. The Receive method is called to process a message received via the Transport Service. Overload this method to add support for processing the message contents and generating any responses. An alternative method for receiving messages is to register a callback with the Transport service (see example programs for how to do this). Finally, the CheckServiceStatus method is called periodically when the Component (explained later) your service belongs to is updated. Add any periodic checking or additional functionality you need to have done repeatedly in this method. An example of how to create your own service is listed in the example_service1.cpp program included with this library.

3.7 COMPONENTS

The Component interface is a collection of services. This class comes with all the Core Services already added by default, and with helper methods to get pointers to them. Before your services will work, they must be added to a Component using the AddService method. However, services can only be added to a component before it is initialized. When a component is initialized, the ID is set, and the Transport layer and other services are initialized also. It is OK to keep pointers to services added to a component for direct access to their data, you just cannot remove them from a Component once they are added.

3.8 CONCLUSION

This section covered the Core Library at a high level, giving basic information for creation of interfaces and use of the library. In order to really understand how to use the library, look at the example programs included with the

library. All example programs are fully commented and simplified to try make it easy for new people to learn. However, you do need to have an understanding of what JAUS is and how the architecture is defined to truly make sense of the library.